NSFileHandle Class Cluster

Class Cluster Description

NSFileHandle objects provide an object-oriented wrapper for accessing open files or communications channels.

The objects you create using this class are called *file handle objects*. Because of the nature of class clusters, file handle objects are not actual instances of the NSFileHandle class but of one of its private subclasses. Although a file handle object's class is private, its interface is public, as declared by the abstract superclass NSFileHandle.

Generally, you instantiate a file handle object by sending one of the **fileHandle...** messages to the NSFileHandle class object. These methods return a file handle object pointing to the appropriate file or communications channel. As a convenience, NSFileHandle provides class methods that create objects representing files and devices in the file system and that return objects representing the standard input, standard output, and standard error devices. You can also create file handle objects from UNIX file descriptors or non-UNIX file handles (particularly Windows HANDLEs) using the **initWithFileDescriptor:** and **initWithNativeHandle:** methods. If you create file handle objects with these last two methods, you "own" the represented descriptor or handle and are responsible for removing it from system tables, usually by sending the file handle object a **closeFile** message .

NSFileHandle

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSFileHandle.h

Class Description

An NSFileHandle is an object that represents an open file or a communications channel. It enables programs to read data from or write data to the represented file or channel. You can use other OpenStep methods for reading from and writing to files—NSFileManager's **contentsAtPath:** and NSData's **writeToFile:atomically:** are but a couple of examples. Why would you use NSFileHandle then? What are its advantages?

- NSFileHandle gives you greater control over input/output operations on files. It allows more manipulatve operations on and within open files, such as seeking, truncating. and reading and writing at an exact position within a file (the file pointer). Other OpenStep methods read or write a file in its entirety; with NSFileHandle, you can range over an open file and insert, extract, and delete data.
- The scope of NSFileHandle is not limited to files. It provides the only OpenStep object that can read and write to communications channels such as those implemented by sockets, pipes, and devices.
- NSFileHandle makes possible asynchronous background communication. With it a program can connect to, and read from, a socket in a separate thread. (See Background Inter-Process Communication Using Sockets below for details on how this is done.)
- NSFileHandle enhances application portability. Its API supports I/O operations on UNIX file systems as well as on certain non-UNIX file sysrems such as Windows NT. Because this API is closer in semantics to traditional UNIX file I/O, it makes it easier for UNIX programmers to port applications.

Note: Instances of NSPipe, a class closely related to NSFileHandle, represent pipes: unidirectional interprocess communication channels that are found on both UNIX systems and on Windows NT. See the NSPipe specification for details.

Background Inter-Process Communication Using Sockets

Sockets are full-duplex communication channels between processes either local to the same host machine or where one process is on a remote host. Unlike pipes, in which data goes in one direction only, sockets allow processes both to send and receive data. NSFileHandle facilitates communciation over stream-type sockets by providing mechanisms run in background threads that accept socket connections and read from sockets.

Note: NSFileHandle currently handles only communciation through stream-type sockets. If you want to use datagrams or other types of sockets, you must create and manage the connection using native system routines.

The process on one end of the communication channel (the server) starts by creating and preparing a socket using system routines. These routines vary slightly between UNIX and non-UNIX systems (such as Windows), but consist of the same sequence of steps:

1. Create a stream-type socket of a certain protocol.
2. Bind a name to the socket.
3. Listen for incoming connections to the socket.

Typically the other process (the client) then locates the named socket created by the first process. Instead of accepting a connection to the socket by calling the appropriate system routine, the client performs the following sequence of steps:

1. It creates an NSFileHandle using the socket identifier as argument to **initWithFileDescriptor:** or **initWithNativeHandle:**, whichever is appropriate to the system on which the socket was created.
2. It adds itself as an observer of NSFileHandleConnectionAcceptedNotification.
3. It sends **acceptConnectionInBackgroundAndNotify** to this NSFileHandle. This method accepts the connection in the background, creates a new NSFileHandle from the new socket descriptor, and posts a NSFileHandleConnectionAcceptedNotification.
4. In a method implemented to respond to this notification, the client extracts the NSFileHandle representing the "near" socket of the connection from the notification's **userInfo** dictionary; it uses the NSFileHandleNotificationFileHandleItem key to do this.

The client can now send data to the other process over the communications channel by sending **writeData:** to the NSFileHandle. (Note that **writeData:** can block.) The client can also read data directly from the NSFileHandle, but this would cause the process to block until the socket connection was closed, so it is usually better to read in the background. To do this, the process must:

5. Add itself as an observer of NSFileHandleReadCompletionNotification or NSFileHandleReadToEndOfFileCompletionNotification.
6. Send **readInBackgroundAndNotify** or **readToEndOfFileInBackgroundAndNotify** to this NSFileHandle. The former method sends a notification after each transmission of data; the latter method accumulates data and sends a notification only after the sending process shuts down its end of the connection.
7. In a method implemented to respond to either of these notifications, the process extracts the transmitted or accumulated data from the notification's **userInfo** dictionary by using the NSFileHandleNotificationDataItem key.

You close the communications channel in both directions by sending **closeFile** to the NSFileHandle; either process can partially or totally close communication across the socket connection with a system-specific shutdown command.

Method Types

Getting an NSFileHandle	+ fileHandleForReadingAtPath: + fileHandleForWritingAtPath: + fileHandleForUpdatingAtPath: + fileHandleWithStandardError + fileHandleWithStandardInput + fileHandleWithStandardOutput + fileHandleWithNullDevice
Creating an NSFileHandle	– initWithFileDescriptor: – initWithFileDescriptor:closeOnDealloc: – initWithNativeHandle: – initWithNativeHandle:closeOnDealloc:
Getting a file descriptor	– fileDescriptor
Getting a native file handle	– nativeHandle
Reading from an NSFileHandle	– availableData – readDataToEndOfFile – readDataOfLength:
Writing to an NSFileHandle	– writeData:
Communicating asynchronously in the background	– acceptConnectionInBackgroundAndNotifyForModes: – acceptConnectionInBackgroundAndNotify – readInBackgroundAndNotifyForModes: – readInBackgroundAndNotify – readToEndOfFileInBackgroundAndNotifyForModes: – readToEndOfFileInBackgroundAndNotify
Seeking within a file	– offsetInFile – seekToEndOfFile – seekToFileOffset:
Operating on a file	– closeFile – synchronizeFile – truncateFileAtOffset:

Class Methods

fileHandleForReadingAtPath:

+ **fileHandleForReadingAtPath:**(NSString *)*path*

Returns an NSFileHandle initialized for reading the file, device, or named socket at *path*. The file pointer is set to the beginning of the file. The returned object responds only to NSFileHandle read messages. If no file exists at *path* the method returns **nil**.

See also: – **availableData**, – **initWithFileDescriptor:**, – **initWithNativeHandle:**, – **readDataOfLength:**, – **readDataToEndOfFile**

fileHandleForUpdatingAtPath:

+ **fileHandleForUpdatingAtPath:**(NSString *)*path*

Returns an NSFileHandle initialized for reading and writing to the file, device, or named socket at *path*. The file pointer is set to the beginning of the file. The returned object responds to both NSFileHandle read messages and **writeData:**. If no file exists at *path* the method returns **nil**.

See also: – **availableData**, – **initWithFileDescriptor:**, – **initWithNativeHandle:**, – **readDataOfLength:**, – **readDataToEndOfFile**

fileHandleForWritingAtPath:

+ **fileHandleForWritingAtPath:**(NSString *)*path*

Returns an NSFileHandle initialized for writing to the file, device, or named socket at *path*. The file pointer is set to the beginning of the file. The returned object responds only to **writeData:**. If no file exists at *path* the method returns **nil**.

See also: – **initWithFileDescriptor:**, – **initWithNativeHandle:**

fileHandleWithNullDevice

+ **fileHandleWithNullDevice**

Returns an NSFileHandle assoicated with a null device. You can use null-device NSFileHandles as "placeholders" for standard-device NSFileHandles or in collection objects to avoid exceptions and other errors resulting from messages being sent to invalid NSFileHandles. Read messages sent to a null-device NSFileHandle return an end-of-file indicator (an empty NSData) rather than raise an exception. Write messages are no-ops whereas **nativeHandle** and **fileDescriptor** return an illegal value (as defined by the underlying operating system). Other methods are no-ops or return "sensible" values.

See also: – **initWithFileDescriptor:**, – **initWithNativeHandle:**

fileHandleWithStandardError

+ **fileHandleWithStandardError**

Returns the NSFileHandle associated with the standard error file, conventionally a terminal device to which error messages are sent. There is one such NSFileHandle per process; it is a shared instance.

See also: + **fileHandleWithNullDevice**, – **initWithFileDescriptor:**

fileHandleWithStandardInput

+ **fileHandleWithStandardInput**

Returns an NSFileHandle associated with the standard input file, conventionally a terminal device on which the user enters a stream of data. There is one such NSFileHandle per process; it is a shared instance.

See also: + **fileHandleWithNullDevice**, – **initWithFileDescriptor:**

fileHandleWithStandardOutput

+ **fileHandleWithStandardOutput**

Returns an NSFileHandle associated with the standard output file, conventionally a terminal device which receives a stream of data from a program. There is one such NSFileHandle per process; it is a shared instance.

See also: + **fileHandleWithNullDevice**, – **initWithFileDescriptor:**

Instance Methods

acceptConnectionInBackgroundAndNotify

– (void)**acceptConnectionInBackgroundAndNotify**

Accepts a socket connection (for stream-type sockets only) in the background and creates a NSFileHandle for the "near" (client) end of the communcations channel. This method is asynchronous. In a separate "safe" thread it accepts a connection, creates an NSFileHandle for the other end of the connection, and returns that object to the client by posting a NSFileHandleConnectionAcceptedNotification in the run loop of the client. The notification includes as data a **userInfo** dictionary containing the created NSFileHandle; access this object using the NSFileHandleNotificationFileHandleItem key.

The receiver must be created by an **initWithFileDescriptor:** or **initWithNativeHandle:** message that takes as an argument a stream-type socket created by the appropriate system routine. The object that will write

data to the returned NSFileHandle must add itself as an observer of NSFileHandleConnectionAcceptedNotification.

See also: – **enqueueNotification:postingStyle:coalesceMask:forModes:** (NSNotificationQueue), – **readInBackgroundAndNotify**, – **readToEndOfFileInBackgroundAndNotify**

acceptConnectionInBackgroundAndNotifyForModes:

– (void)**acceptConnectionInBackgroundAndNotifyForModes:**(NSArray *)*modes*

Asynchronously accepts a connection with a stream-type socket in the background and returns (in an NSFileHandleConnectionAcceptedNotification) an NSFileHandle representing the client side of the connection. See **acceptConnectionInBackgroundAndNotify** for details. The method differs from **acceptConnectionInBackgroundAndNotify** in that *modes* specifies the run-loop mode (or modes) in which NSFileHandleConnectionAcceptedNotification can be posted.

See also: – **enqueueNotification:postingStyle:coalesceMask:forModes:** (NSNotificationQueue), – **readInBackgroundAndNotifyForModes:**, – **readToEndOfFileInBackgroundAndNotifyForModes:**

availableData

– (NSData *)**availableData**

If the receiver is a file, returns the data obtained by reading the file from the file pointer to the end of the file. If the receiver is a communications channel, reads up to a buffer of data and returns it (the size of the buffer depends on the operating system); if no data is available, the method blocks. Returns an empty NSData if the end of file is reached. Raises NSFileHandleOperationException if attempts to determine file-handle type fail or if attempts to read from the file or channel fail.

See also: – **readDataOfLength:**, – **readDataToEndOfFile:**

closeFile

– (void)**closeFile**

Disallows further access to the represented file or communications channel and signals end of file on communications channels that permit writing. The file or communications channel, however, is available for other uses. Further read and write messages sent to an NSFileHandle to which **closeFile** has been sent will raise an exception.

Note: Sending **closeFile** to an NSFileHandle does not cause its deallocation; for that, you must send it **release**. Deallocation of an NSFileHandle, on the other hand, deletes its descriptor and closes the represented file or channel. Use **closeFile** when you want to close a file immediately and reclaim the descriptor; use **release** when it's acceptable to defer this.

fileDescriptor

– (int)**fileDescriptor**

Returns the file descriptor associated with the receiver. You can send this message to NSFileHandles of both UNIX and non-UNIX origin and receive a valid file descriptor (unless **closeFile** has been sent to the object, in which case an exception is raised).

See also: – **initWithFileDescriptor:**

initWithFileDescriptor:

– (id)**initWithFileDescriptor:**(int)*fileDescriptor*

Returns an NSFileHandle initialized with the file *descriptor.* If the operating system is Windows, the method converts the file descriptor to a file handle (WIN32 type HANDLE) and initializes the returned object with that as well. You can create an NSFileHandle for a socket on a UNIX system by using the result of a **socket** call as *descriptor.* The object creating an NSFileHandle using this method owns *fileDescriptor* and is responsible for its disposition.

See also: – **closeFile**

initWithFileDescriptor:closeOnDealloc:

– (id)**initWithFileDescriptor:**(int)*fileDescriptor* **closeOnDealloc:**(BOOL)*flag*

Same as **initWithFileDescriptor:**, but *flag*, if YES, causes the file descriptor to be closed when the receiver is deallocated.

See also: – **closeFile**

initWithNativeHandle:

– (id)**initWithNativeHandle:**(void *)*handle*

Returns an NSFileHandle initialized with a file *handle* from a non-UNIX system (currently only a handle of WIN32's HANDLE type). The method converts the handle to a Windows C run-time file descriptor and initializes the returned NSFileHandle with that as well. If a problem occurs converting the handle to a descriptor, the method returns **nil**. The object creating an NSFileHandle using this method owns *fileDescriptor* and is responsible for its disposition.

See also: – **closeFile**

initWithNativeHandle:closeOnDealloc

– (id)**initWithNativeHandle:**(void *)*handle* **closeOnDealloc:**(BOOL)*flag*

Same as **initWithNativeHandle:**, but *flag*, if YES, causes the handle to be closed when the receiver is deallocated.

See also: – **closeFile**

nativeHandle

– (void *)**nativeHandle**

On non-UNIX operating systems (particularly Windows), returns the file handle associated with the receiver. On UNIX systems, returns a NULL **void *** pointer.

See also: – **initWithNativeHandle:**

offsetInFile

– (unsigned long long)**offsetInFile**

Returns the position of the file pointer within the file represented by the receiver. Raises an exception if the message is sent to an NSFileHandle representing a pipe or socket or if the file descriptor is closed.

See also: – **seekToEndOfFile**, – **seekToFileOffset:**

readDataOfLength:

– (NSData *)**readDataOfLength:**(unsigned int)*length*

If the receiver is a file, returns the data obtained by reading from the file pointer to *length* or to the end of the file, whichever comes first. If the receiver is a communications channel, the method reads data from the channel up to *length*. Returns an empty NSData if the file is positioned at the end of the file or if an end-of-file indicator is returned on a communications channel. Raises NSFileHandleOperationException if attempts to determine file-handle type fail or if attempts to read from the file or channel fail.

See also: – **availableData**, – **readDataToEndOfFile:**

readDataToEndOfFile

– (NSData *)**readDataToEndFile**

Invokes **readDataOfLength:**, reading up to UNIT_MAX bytes (the maximum value for unsigned integers) or, if a communications channel, until an end-of-file indicator is returned.

See also: – **availableData**

readInBackgroundAndNotify

– (void)**readInBackgroundAndNotify**

Performs an asynchronous **availableData** operation on a file or communications channel and posts an NSFileHandleReadCompletionNotification to the client process' run loop. The length of the data is limited to the buffer size of the underlying operating system. The notification includes a **userInfo** dictionary which contains the data read; access this object using the NSFileHandleNotificationDataItem key.

Any object interested in receiving this data asynchronously must add itself as an observer of NSFileHandleReadCompletionNotification. In communication via stream-type sockets, the receiver is often the object returned in the **userInfo** dictionary of NSFileHandleConnectionAcceptedNotification.

See also: – **acceptConnectionInBackgroundAndNotify**,
– **enqueueNotification:postingStyle:coalesceMask:forModes:** (NSNotificationQueue),

readInBackgroundAndNotifyForModes:

– (void)**readInBackgroundAndNotifyForModes:**(NSArray *)*modes*

Performs an asynchronous **availableData** operation on a file or communications channel and posts an NSFileHandleReadCompletionNotification to the client process' run loop. See **readInBackgroundAndNotify** for details. The method differs from **readInBackgroundAndNotify** in that *modes* specifies the run-loop mode (or modes) in which NSFileHandleReadCompletionNotification can be posted.

See also: – **acceptConnectionInBackgroundAndNotifyforModes:**,
– **enqueueNotification:postingStyle:coalesceMask:forModes:** (NSNotificationQueue),

readToEndOfFileInBackgroundAndNotify

– (void)**readToEndOfFileInBackgroundAndNotify**

Performs an asynchronous **readToEndOfFile** operation on a file or communications channel and posts an NSFileHandleReadToEndOfFileCompletionNotification to the client process' run loop. The notification includes a **userInfo** dictionary which contains the data read; access this object using the NSFileHandleNotificationDataItem key.

Any object interested in receiving this data asynchronously must add itself as an observer of NSFileHandleReadToEndOfFileCompletionNotification. In communication via stream-type sockets, the receiver is often the object returned in the **userInfo** dictionary of NSFileHandleConnectionAcceptedNotification.

See also: – **acceptConnectionInBackgroundAndNotify**,
– **enqueueNotification:postingStyle:coalesceMask:forModes:** (NSNotificationQueue)

readToEndOfFileInBackgroundAndNotifyForModes:

– (void)**readToEndOfFileInBackgroundAndNotifyForModes:**(NSArray *)*modes*

In a detached "safe" thread, continuously reads data made available across a socket connection and accumulates it until the connection is closed; it then notifies observers. See **readToEndOfFileInBackgroundAndNotify** for details. The method differs from **readToEndOfFileInBackgroundAndNotify** in that *modes* specifies the run-loop mode (or modes) in which NSFileHandleReadToEndOfFileCompletionNotification can be posted.

See also: – **acceptConnectionInBackgroundAndNotifyforModes:**,
– **enqueueNotification:postingStyle:coalesceMask:forModes:** (NSNotificationQueue)

seekToEndOfFile

– (unsigned long long)**seekToEndOfFile**

Puts the file pointer at the end of the file referenced by the receiver and returns the new file offset (thus yielding the size of the file). Raises an exception if the message is sent to an NSFileHandle representing a pipe or socket or if the file descriptor is closed.

See also: – **offsetInFile**

seekToFileOffset:

– (void)**seekToFileOffset:**(unsigned long long)*offset*

Moves the file pointer to the specified *offset* within the file represented by the receiver. Raises an exception if the message is sent to an NSFileHandle representing a pipe or socket, if the file descriptor is closed, or if any other error occurs in seeking.

See also: – **offsetInFile**

synchronizeFile

– (void)**synchronizeFile**

Causes all in-memory data and attributes of the file represented by the receiver to be written to permanent storage. This method should be invoked by programs that require the file to always be in a known state. An invocation of this method does not return until memory is flushed; because the way memory is flushed is platform-specific, consult the documentation for your operating system.

truncateFileAtOffset:

– (void)**truncateFileAtOffset:**(unsigned long long)*offset*

Truncates or extends the file represented by the receiver to *offset* within the file and puts the file pointer at that position. It the file is extended, the added characters are null bytes.

writeData:

– (void)**writeData:**(NSData *)*data*

Synchronously writes *data* to the file, device, pipe, or socket represented by the receiver. If the receiver is a file, writing takes place at the file pointer's current position. After it writes the data, the method advances the file pointer by the number of bytes written. Raises an exception if the file descriptor is closed or is not valid, if the receiver represents an unconnected pipe or socket end point, if no free space is left on the file system, or if any other writing error occurs.

See also: – **availableData**, – **readDataOfLength:**, – **readDataToEndOfFile**

Notifications

NSFileHandle posts several notifications related to asynchronous background I/O operations. They are set to post when the run loop of the thread that started the asynchronous operation is idle and for a specified set of run-loop modes (see **userInfo** dictionary).

NSFileHandleConnectionAcceptedNotification

Notification Object The posting NSFileHandle

userInfo

Key	Value
NSFileHandleNotificationFileHandleItem	The NSFileHandle representing the "near" end of a socket connection.
NSFileHandleNotificationMonitorModes	An NSArray containing the run-loop modes in which the notification can be posted.

To cause the posting of this notification, you must send either **acceptConnectionInBackgroundAndNotify** or **acceptConnectionInBackgroundAndNotifyForModes:** to an NSFileHandle representing a server stream-type socket. This notification is posted when NSFileHandle establishes a socket connection between two processes, creates an NSFileHandle for one end of the connection, and makes this object available to observers by putting it in the **userInfo** dictionary.

NSFileHandleReadCompletionNotification

Notification Object The posting NSFileHandle

userInfo

Key	Value
NSFileHandleNotificationDataItem	An NSData containing the available data read from a socket connection.
NSFileHandleNotificationMonitorModes	An NSArray containing the run-loop modes in which the notification can be posted.

To cause the posting of this notification, you must send either **readInBackgroundAndNotify** or **readInBackgroundAndNotifyForModes:** to an appropriate NSFileHandle. This notification is posted when the background thread reads the data currently available in a file or at a communications channel. It makes the data available to observers by putting it in the **userInfo** dictionary.

NSFileHandleReadToEndOfFileCompletionNotification

Notification Object The posting NSFileHandle

userInfo

Key	Value
NSFileHandleNotificationDataItem	An NSData containing the available data read from a socket connection.
NSFileHandleNotificationMonitorModes	An NSArray containing the run-loop modes in which the notification can be posted.

To cause the posting of this notification, you must send either **readToEndOfFileInBackgroundAndNotify** or **readToEndOfFileInBackgroundAndNotifyForModes:** to an appropriate NSFileHandle. This notification is posted when the background thread reads all data in the file or, if a communications channel, until the other process signals end of data. It makes the data available to observers by putting it in the **userInfo** dictionary.