Portable Systems Group

NT OS/2 Named Pipe Specification

Author: *David N. Cutler & Gary D. Kimura*

Original Draft February 16, 1990
Revision 1.1, March 8, 1990
Revision 1.2, August 14, 1990
Revision 1.3, September 27, 1990
Revision 1.4, October 17, 1990
Revision 1.5, January 23, 1991

1. Introduction

This specification discusses the named pipe facilities of **NT OS/2**. Named pipes provide a full duplex interprocess communication (IPC) mechanism that can be used locally or across a network to access application servers. Named pipes provide the transport medium that is used for the Microsoft remote procedure call (RPC) capabilities.

Named pipes are used extensively by the OS/2 and LAN Manager components of the **NT OS/2** system, and therefore, must be implemented as efficiently as possible.

There are two manifestations of named pipes, those that are local to a system and those that are remote. This specification addresses both types of named pipes.

In addition to describing the **NT OS/2** named pipe facilities, this specification also discusses the way in which the OS/2 named pipe APIs are emulated.

2. Goals

The major goals for the named pipe capabilities of **NT OS/2** are the following:

1. Provide the basic primitives necessary to compatibly emulate the OS/2 named pipe capabilities.

2. Provide protection and security attributes for named pipes that are comparable to the capabilities provided for files and other **NT OS/2** objects.

3. Provide for LAN Manager server and client redirection of named pipes without having to enter the OS/2 subsystem.

4. Provide a fully qualified name space for named pipes that fits into the **NT OS/2** name structure in a straightforward manner.

5. Provide a high performance design and implementation of named pipes.

Although it is a major temptation, it is not a goal to "fix" the semantics of OS/2 named pipes. Minor discrepancies, however, will exist between OS/2 and **NT OS/2** named pipes where OS/2 capabilities or semantics are incompatible with those of **NT OS/2**, e.g., the named pipe naming and the asynchronous I/O model.

3. Overview of OS/2 Named Pipes

A named pipe provides a full duplex channel that can be used to implement an interprocess communication (IPC) mechanism between two processes. OS/2 uses named pipes to implement location-independent remote procedure call (RPC) capabilities and for communicating with servers on a remote system.

Named pipes have two ends: 1) a client end, and 2) a server end. Both ends are full duplex—data written from one end can be read from the other end and vice versa.

The server end of a named pipe is created when a new instance of a named pipe is created, or when a previously created instance is reused. A new instance of a named pipe is created with the **DosMakeNmPipe** API in OS/2.

Before either the client or the server ends of a named pipe can be used, the server end must be connected. In OS/2 this is accomplished with the **DosConnectNmPipe** API.

Once an instance of a named pipe is created and the server end is connected, then the client end of the named pipe can be created using the OS/2 **DosOpen** API.

When both the server end of a named pipe is connected and the client end is opened, information can flow over the pipe using the OS/2 **DosRead** and **DosWrite** APIs.

Named pipes are created with five attributes:

1. A pipe type which is either message or byte stream.

2. A count that limits the maximum number of simultaneous instances of the named pipe that can be created.

3. An input buffer size that specifies the size of the buffer that is used for inbound data on the server side of the named pipe.

4. An output buffer size that specifies the size of the buffer that is used for outbound data from the server side of the named pipe.

5. A default timeout value that is to be used if a timeout value is not specified when the **DosWaitNmPipe** API is executed.

The type of a named pipe determines how information is written into the named pipe. If the named pipe is a message pipe, then information is written into the pipe in the form of messages which include the byte count and the data of the message. If the named pipe is a byte stream pipe, then only the data is written into the named pipe.

The maximum instance count is established when the first instance of a specific named pipe is created (i.e., one of a given name) and cannot later be modified. Thereafter, up to the maximum instance count of simultaneous instances of the named pipe can be created to provide an IPC mechanism between any pair of processes.

The input and output buffer sizes are considered hints to the system for the sizes of the buffers that are needed to buffer inbound and outbound data. The actual buffer sizes may be either the system default or the specified buffer sizes rounded up to the next allocation boundary.

The default timeout value specifies a default for the amount of time that a client can wait for an available instance of a named pipe.

Once the first instance of a named pipe is created subsequent instances of an identically named pipe are subject to the maximum instances parameter. In addition, the type of pipe and the default timeout value are ignored and cannot be set when subsequent instances of the named pipe are created.

In addition to the five attribute parameters, two mode parameters can be specified when an instance of a named pipe is created or opened:

1. The read mode, which can be either message mode or byte stream mode, but which must be compatible with the type of the named pipe.

2. The blocking mode, which can be either blocking or nonblocking.

The read mode of a named pipe determines how data will be read from the pipe. If the named pipe is a message pipe, then data can be read in either message mode or byte stream mode. However, if the named pipe is a byte stream pipe, then data can only be read in byte stream mode.

The blocking mode determines what happens when a request cannot be satisfied immediately. If the mode is blocking, then an implied wait occurs until an operation is completed. Otherwise, the operation returns immediately with an error status.

Standard open parameters can also be specified when an instance of a named pipe is created or opened which define the access that is desired to the named pipe (e.g., read only, write only, or read/write access), whether the named pipe handle is inherited when a child process is created, and whether write behind is allowed on writes to the named pipe.

The open access parameters also specify the configuration of the named pipe when the first instance of a named pipe is created. A named pipe can have a full duplex or a simplex configuration. A full duplex named pipe allows data to flow in both directions, whereas a simplex named pipe only allows data to flow in one direction. The direction of data flow and configuration are determined by the read only (outbound), write only (inbound), and read/write (full duplex) open access parameters specified by the server when the first instance of a named pipe is created.

The server end of a named pipe can be reused by disconnecting the client end. In OS/2, this is accomplished using the **DosDisconnectNmPipe** API. The server end of a named pipe can also be disconnected by closing the respective file handle, but this deletes the instance of the named pipe and it cannot be reused.

The client end of a named pipe is disconnected by simply closing the respective file handle.

OS/2 supplies 14 APIs that are specific to named pipes. These APIs are intended mainly for use by a server. In addition, eleven standard OS/2 I/O system APIs can be executed using a file handle to a named pipe.

4. Overview of NT OS/2 Named Pipes

4.1 Implementation Alternatives

Named pipes must be integrated into the **NT OS/2** I/O system such that standard read and write requests can be used to read data from and write data to a named pipe. It also must be possible to accomplish LAN Manager server and client redirection of named pipes without having to call the OS/2 subsystem.

There are several ways of integrating named pipes into **NT OS/2** that meet these requirements:

1. Implement the named pipe capabilities as an installable file system and extend **NtCreateFile** so that the named pipe attributes required by OS/2 can be specified directly in the **NT OS/2** system service call.

2. Implement the named pipe capabilities as an installable file system and use extended attributes as the means of defining the named pipe attributes required by OS/2.

3. Implement named pipes as a separate object that is created with its own API, but which can be opened via a pipe driver.

4. Implement the named pipe capabilities as an installable file system and add an **NT OS/2** I/O system API that specifically creates an instance of a named pipe.

The first alternative requires an already complicated API to be further extended to accommodate yet another special case.

The second alternative overloads the use of extended attributes to have a special meaning for named pipes. Extended attributes are not the most efficient or convenient way of specifying the attribute values and would require special rules about when they could be read and written.

The third alternative would create a nonstandard object whose API was partly buried in the I/O system and partly in object-specific APIs.

The fourth alternative adds an additional API to the **NT OS/2** I/O system that has special meaning and is only applicable to named pipes.

The fourth alternative has been chosen as the means of implementing the named pipe capabilities in **NT OS/2**. Although this provides an additional I/O system API that is specific to named pipes, it is the most straightforward and efficient implementation.

4.2 Named Pipe Directories

In OS/2, named pipes have a rigid name syntax with the following form:

\PIPE\pipe-name

This syntax is recognized by the OS/2 **DosOpen** API and is routed to the appropriate system component. The LAN Manager redirector is also capable of recognizing names of the following form*:*

*server-name**PIPE**pipe-name*

The redirector transforms the request into a tree connection to a server and then performs the appropriate SMB generation.

The **NT OS/2** named pipe driver will also implement a flat name space. The syntax for an **NT OS/2** named pipe is of the following form:

*Device**NamedPipe**pipe-name*[1]

*\The object name space in **NT OS/2** is more general and hierarchical, and we would like named pipes to follow that scheme; however, because of issues involving persistent named pipes, and guaranteeing proper behavior given reparse the first named pipe driver will use a flat name space. Once the issues are resolved named pipes can be extended to existing file systems as a special file using reparse or by maintaining a named pipe database in a system file.*

The syntax for a remote **NT OS/2** named pipe is of the following form:

*Device**LanmanRedirector**server-name**Pipe**pipe-name*

4.3 Read/Write Buffering Strategy

4.3.1 OS/2 Read/Write Buffering Strategy

The OS/2 named pipe capabilities use a two circular buffers for buffering inbound and outbound writes to a named pipe. This design is dictated by the synchronous I/O model of OS/2 and it controls the amount of system buffering space that is consumed. The data is copied twice for each write and read of a named pipe. One copy occurs when the data is written from a user buffer into a named pipe and another copy occurs when the data is read out of the named pipe into a user buffer.

An OS/2 named pipe can be either a message or byte stream pipe, which determines how write data is stored in the pipe buffers. Message pipes can be read in either message mode or byte stream mode. Byte stream pipes can only be read in byte stream mode. In addition, a blocking mode can be specified for each open of an instance of a named pipe. The blocking mode determines whether reads from, and writes to the named pipe block if sufficient data or space is not available in the named pipe.

---

[1]The string "\Device\NamedPipe" refers to the named pipe driver, while the string "\Device\NamedPipe\" represents the root directory of the named pipe file system.

A message named pipe stores the size of a message and the data for the message. A byte stream named pipe simply stores the data and no additional information. Reads from a message named pipe attempt to read a complete message from the pipe in either message mode or byte stream mode. If the complete message does not fit in the supplied read buffer, then a full buffer is returned along with an error status that signifies that there is more data in the message. Reads from a byte stream named pipe can only be made in byte stream mode and return the data that is currently in the pipe up to the size of the supplied buffer.

Each inbound and outbound buffer for a named pipe has a read lock, a write lock, a read semaphore, and a write semaphore. These are used to synchronize the reading and writing of data to and from the buffer.

When a write to a named pipe buffer begins, the write lock is acquired to prevent any other writer from writing into the buffer until the current write is finished. If the write must block because of a lack of available space in the buffer, then the reader semaphore is signaled, the writer lock is not released, and the writer waits for a reader to signal the write semaphore. The write lock is released at the completion of the write operation.

The following describes the OS/2 named pipe write logic.

```
if (message pipe) then
        if (blocking mode) then
                write message to pipe, synchronize with reader
                return size of message written
        else
                if (space for message header plus one byte) and
                  (data buffer size greater than pipe buffer size) then
                        write data to pipe, synchronize with reader
                        return size of message written
                else
                        if (space for data buffer and message header) then
                                write data to pipe
                                return size of message written
                        else
                                return buffer overflow error
                        endif
                endif
        endif
else
        if (blocking mode) then
                write data to pipe, synchronize with reader
                return count of bytes written
        else
                if (space available in pipe buffer) then
                        write data to pipe (minimum data buffer/pipe space)
                        return count of bytes written
                else
                        return buffer overflow error
                endif
        endif
endif
```

When a read from a named pipe buffer begins, the read lock is acquired to prevent any other reader from reading from the buffer until the current read is finished. If the read must block because of a lack of available data in the buffer, then the writer semaphore is signaled, the reader lock is not released, and the reader waits for the read semaphore to be signaled. The read lock is released at the completion of the read operation.

The following describes the OS/2 named pipe read logic.

```
while (data not available in pipe) do
        if (blocking mode) then
                wait for available data in pipe
        else
                return no data available error
        endif
endwhile

if (message pipe) then
        if (data buffer size greater or equal message size) then
                if (message mode) then
                        read message from pipe, synchronize with writer
                        return size of message read
                else
                        read available data or message from pipe
                        if (complete message read) then
                                return size of message read
                        else
                                reduce size of message by available data bytes
                                return count of data bytes read
                        endif
                endif
        else
                if (message mode) then
                        read data from pipe, synchronize with writer
                        reduce size of message by data buffer size
                        return more data error
                else
                        read available data from pipe
                        reduce size of message by available data bytes
                        return count of data bytes read
                endif
        endif
else
        read available data from pipe
        return count of data bytes read
endif
```

4.3.2 NT OS/2 Read/Write Buffering Strategy

NT OS/2 supports an asynchronous I/O model and uses the concept of quotas to control the allocation of system buffers. In addition, **NT OS/2** supports I/O transfers that are buffered by the system rather than requiring buffers to be locked down and nonswappable. Therefore, the buffering scheme used for the **NT OS/2** implementation of named pipes differs markedly from that of OS/2.

The blocking mode of OS/2 is emulated in **NT OS/2** with a completion mode. The completion mode can be specified such that read and write operations are completed immediately or they are queued and subject to completion when space is available or data is present.

The inbound and outbound buffers for a named pipe are not actually allocated to real memory in **NT OS/2**. Instead, the creator of an instance of a named pipe is simply charged memory quota for these buffers. Writers and readers can use up to the quota charged to the creator without having any quota charged against themselves. If the quota charged to the creator is exhausted and a write request is queued rather than completed immediately, then the writer is charged for any additional quota that is required. Likewise, a reader is charged quota if no data is available, the quota charged to the creator is exhausted, and the read request is queued rather than completed immediately.

The named pipe capabilities of **NT OS/2** requires that the data be copied twice. However, reads can "pull" data from write buffers and writes can "push" data into read buffers.

The following is a somewhat simplified discussion of the buffering scheme used for named pipes in **NT OS/2**. Boundary conditions and differing pipe types and modes are not considered. The pipe type is assumed to be message, the read mode is assumed to be message, and the completion mode is assumed to be queued operation.

The exact behavior of the **NT OS/2** named pipe buffering depends on whether a read occurs before a write or vice versa.

If a write operation occurs before a read operation, then the writer's output buffer is probed for read accessibility in the requesting mode. A system buffer is allocated that is the required size to hold the write data and memory quota is charged to the writer if and only if the quota charged to the creator of the named pipe instance has been exhausted (e.g., because of a previous read or write request). A buffer header is initialized at the front of the system buffer, the write data is copied into the system buffer, and the buffer header is inserted into the first-in-first-out list of writers. If quota was not charged to the writer, then the writer's I/O request can be completed immediately. The system buffer will be deallocated and the creator's quota returned when a matching read arrives. Otherwise, the write request type is converted to a buffered request so that upon completion, the I/O system will deallocate the system buffer and return memory quota as appropriate.

At this point, the I/O operation is either pending or has been completed and control is returned to the caller. If another write request is received before the first set of write data is read, then the same operations are performed and the new request is placed at the end of the pending queue.

At some subsequent point in time, a read request arrives at the read end of the pipe and it is determined that write data is available at the write end of the pipe. The input buffer is probed for write accessibility in the requesting mode. The read then proceeds to "pull" (copy) data directly from the system buffer that was previously allocated for the write data into the user's input buffer. At the completion of the copy, the read I/O request is completed.

Completion of the read request involves writing the I/O status block and setting the completion event. If the original write I/O request was completed at the time of the write, then the system buffer is deallocated and memory quota is returned for named pipe write buffering. However, if the write I/O was not completed at the time of the write, then completion of the write requires writing the I/O status block, setting the completion event, deallocating the system buffer, and returning quota to the writer.

If an access violation occurs during a copy from the output buffer to a system buffer, then the write operation is immediately terminated. Previously completed read I/O requests, if any, are not backed out. This has no effect on the integrity of the system. A malicious writer could easily accomplish the same effect by simply writing a shortened message. The write I/O status is set to access violation, the write I/O request is completed, and successful completion is returned as the service status.

The following describes the **NT OS/2** named pipe write logic.

```
probe output buffer for read access
while (read pending) and (output buffer size not zero) do
        copy data from output buffer to read buffer
        if (read buffer greater or equal output buffer) then
                reduce output buffer size to zero
                set read I/O status to successful completion
        else
                reduce output buffer by read buffer size
                if (message mode read) then
                        set read I/O status to buffer overflow
                else
                        set read I/O status to successful completion
                endif
        endif
        remove read request from read pending list
        complete read I/O operation, return quota
endwhile
if (output buffer size not zero) then
        if (pipe quota available) then
                allocate write buffer, charge quota to pipe
                copy data from output buffer to write buffer
                insert write request in write pending list
                set write I/O status to successful completion
                complete write I/O operation
                return successful completion
        else
                if (queued operation) or ((message pipe) and
                        (output buffer not original size)) then
                        allocate write buffer, charge quota to writer
                        copy data from output buffer to write buffer
                        insert write request in write pending list
                        return operation pending
                else
                        if (output buffer not original size) then
                                set write I/O status to successful completion
                                complete write I/O request
                                return successful completion
                        else
                                abort write I/O operation
                                return no space available
                        endif
                endif
        endif
else
        set write I/O status to successful completion
        complete write I/O request
        return successful completion
endif
```

If a read operation occurs before a write, then the reader's input buffer is probed for write accessibility in the requesting mode. A system buffer is allocated that is the required size to hold the input data and memory quota is charged to the reader if and only if the quota charged to the creator of the named pipe instance has been exhausted (e.g., because of a previous read or write request). A buffer header is initialized at the front of the system buffer and the header is inserted in a first-in-first-out list of readers. The read request type is converted to a buffered request so that upon completion, the I/O system will copy the received data from the system buffer into the reader's input buffer, deallocate the system buffer, and return memory quota as appropriate.

At this point, the I/O operation is pending and control is returned to the caller. If another read request is received before the first read is completed, then the same operations are performed and the new request is placed at the end of the pending queue.

At some subsequent point in time, a write request arrives at the write end of the pipe and it is determined that a read is pending at the read end of the pipe. The output buffer is probed for read accessibility in the requesting mode. The write then proceeds to "push" (copy) data directly from the output buffer into the system buffer that was previously allocated for the read operation. At the completion of the copy, the read and write I/O requests are both completed.

Completion of the write request involves writing the I/O status block and setting the completion event, whereas completion of the read request requires copying the read data from the system buffer to the reader's input buffer, deallocating the system buffer and returning the memory quota as appropriate, writing the I/O status block, and setting the completion event.

If an access violation occurs during a copy from a system buffer to the input buffer, then the read operation is immediately terminated. Previously completed write I/O requests are not backed out. This has no effect on the integrity of the system. A malicious reader could easily accomplish the same effect by simply reading and discarding information. The read I/O status is set to access violation, the read I/O request is completed, and successful completion is returned as the service status.

The following describes the **NT OS/2** named pipe read logic.

```
probe input buffer for write access
if (write not pending) then
        if (queued operation) then
                if (pipe quota available) then
                        allocate read buffer,charge quota to pipe
                else
                        allocate read buffer, charge quota to reader
                endif
                insert read request in read pending list
                return operation pending
        else
                abort read I/O operation
                return no data available
        endif
else
        set read I/O status to successful completion
        while (write pending) and (input buffer size not zero) do
                copy data from write buffer to input buffer
                if (input buffer greater or equal write buffer) then
                        if (message mode read) then
                                reduce input buffer size to zero
                        else
                                reduce input buffer by write buffer size
                        endif
                        set write I/O status to successful completion
                        remove write request from write pending list
                        complete write I/O operation, return quota
                else
                        reduce write buffer by input buffer size
                        reduce input buffer size to zero
                        if (message mode read) then
                                set read I/O status to buffer overflow
                        endif
                endif
        endwhile

        complete read I/O operation
        return successful completion
endif
```

4.4 Internal Read/Write Operations

4.4.1 Special Read/Write Buffering

In addition to the above buffering method provided for local named pipe clients and servers, **NT OS/2** provides another buffering method that can be used internally by the **NT OS/2** LAN Manager server. This method allows read and write requests to proceed such that no buffer allocation is needed for either the read or the write.

The **NT OS/2** LAN Manager server supplies the necessary system buffers directly and only one copy of the data is needed for a read or write operation. Typically, these buffers are the buffers that are used to receive and transmit data over the network. Thus, server side redirection can be performed with minimal overhead.

4.5 Named Pipe States

Named pipes can be in one of four states:

1. Disconnected
2. Listening
3. Connected
4. Closing

The initial state of a named pipe is *disconnected*. When the pipe is in this state, no client is connected to the pipe and a listen operation can be performed.

Performing a listen operation on a disconnected named pipe causes the pipe to transition to the *listening* state.

An open request performed by a client causes a named pipe in the listening state to enter the *connected* state. When a named pipe is in the connected state, data can flow through the pipe.

A named pipe that is in the connected state can transition to either the *disconnected* state or the *closing* state.

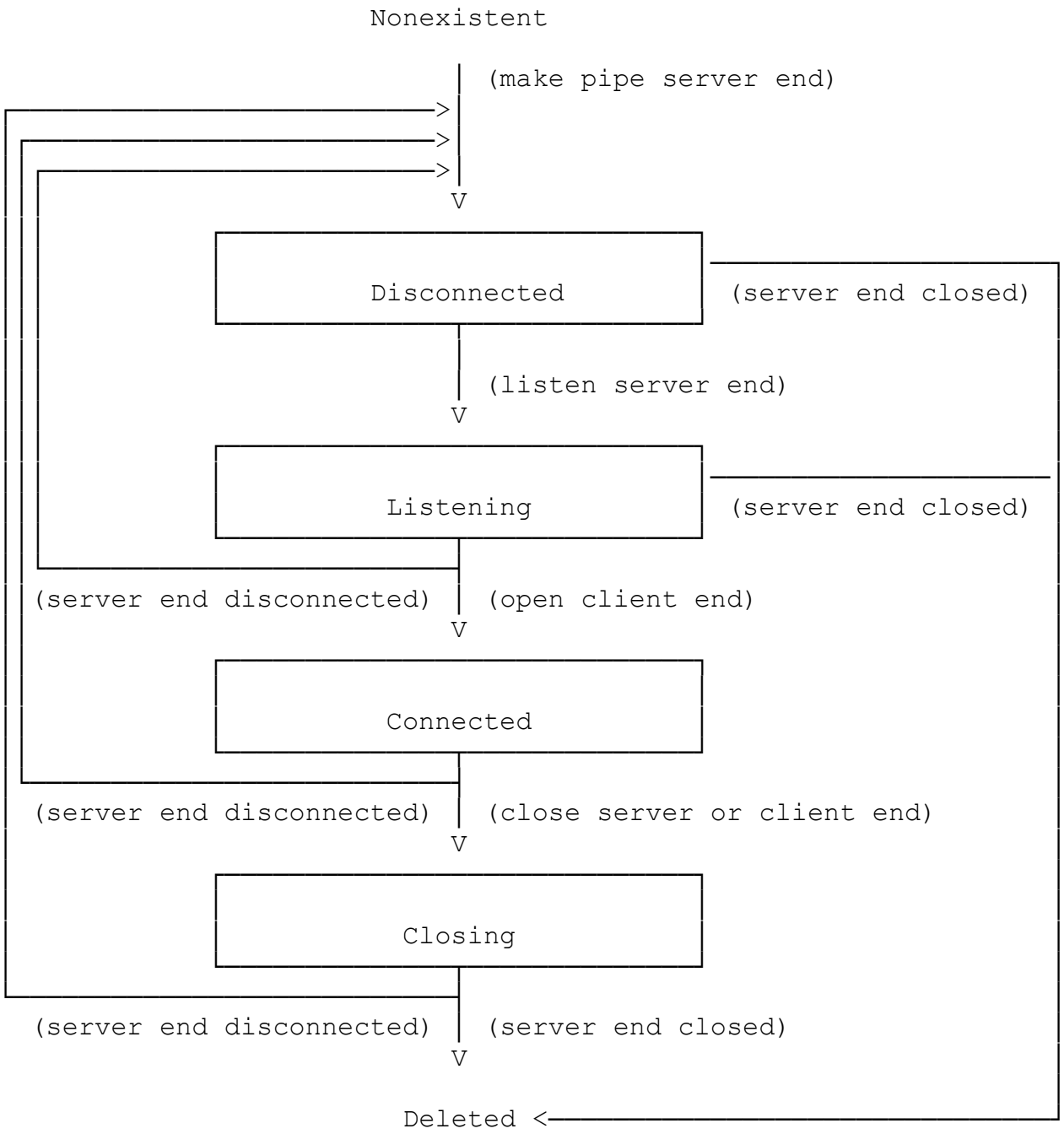
The disconnected state is entered when a disconnect operation is performed on the server end of a named pipe and causes both the input buffer and the output buffer to be flushed. No further access is allowed to the client end of the named pipe; however, the client end must still be closed.

The closing state is entered if a close operation is performed on either end of a named pipe and causes the input buffer of the closing end to be flushed. Any remaining data in the output buffer can be read from the opposite end of the named pipe with a read operation. When no data remains in the output buffer, an end of file indication is returned.

A named pipe that is in the closing state because the client end of the pipe was closed can transition to the disconnected state by performing a disconnect operation on the server end of the pipe.

A named pipe that is in the closing state because the server end of the pipe was closed is deleted when the client end of the pipe is also closed.

Named Pipe State Transition Diagram



5. NT OS/2 Named Pipe I/O Operations

The following subsections describe the **NT OS/2** I/O operations with respect to named pipes. Additional information can be found in the **NT OS/2** I/O System Specification.

5.1 Create Named Pipe

The first instance of a specific named pipe or another instance of an existing named pipe can be created, and a server end handle opened with the **NtCreateNamedPipeFile** function. This function is only for creating local named pipes and not remote ones.

NTSTATUS
NtCreateNamedPipeFile (
    OUT PHANDLE *FileHandle,*
    IN ULONG *DesiredAccess,*
    IN POBJECT_ATTRIBUTES *ObjectAttributes,*
    OUT PIO_STATUS_BLOCK *IoStatusBlock,*
    IN ULONG *ShareAccess,*
    IN ULONG *CreateDisposition,*
    IN ULONG *CreateOptions,*
    IN ULONG *NamedPipeType,*
    IN ULONG *ReadMode,*
    IN ULONG *CompletionMode,*
    IN ULONG *MaximumInstances,*
    IN ULONG *InboundQuota,*
    IN ULONG *OutboundQuota,*
    IN PTIME *DefaultTimeout* **OPTIONAL**
    );

Parameters:

FileHandle - A pointer to a variable that receives the file handle value.

DesiredAccess - Specifies the type of access that the caller requires to the named pipe.

DesiredAccess Flags:

SYNCHRONIZE − The file handle may be waited on to synchronize with the completion of I/O operations.

READ_CONTROL − The ACL and ownership information associated with the named pipe may be read.

WRITE_DAC − The discretionary ACL associated with the named pipe may be written.

WRITE_OWNER − Ownership information associated with the named pipe may be written.

FILE_READ_DATA − Data may be read from the named pipe.

FILE_WRITE_DATA − Data may be written to the named pipe.

FILE_CREATE_PIPE_INSTANCE - This access is needed to create subsequent instances of the named pipe.

FILE_READ_ATTRIBUTES - Named pipe attributes flags may be read.

FILE_WRITE_ATTRIBUTES - Named pipe attribute flags may be written.

The three following values are the generic access types that the caller may request along with their mapping to specific access rights:

GENERIC_READ - Maps to *FILE_READ_DATA* and *FILE_READ_ATTRIBUTES*.

GENERIC_WRITE - Maps to *FILE_WRITE_DATA* and *FILE_WRITE_ATTRIBUTES*.

GENERIC_EXECUTE - Maps to *SYNCHRONIZE*.

ObjectAttributes − A pointer to a structure that specifies the object attributes; refer to the I/O System Specification for details.

IoStatusBlock - A pointer to a structure that receives the final completion status. The actual action taken by the system is written into the *Information* field of this structure. For example, it indicates if a new named pipe and instance was created or just a new instance.

ShareAccess - Specifies the share access and configuration of the named pipe.

ShareAccess Flags:

FILE_SHARE_READ - Indicates that client end handles can be opened for read access to the named pipe.

FILE_SHARE_WRITE - Indicates that client end handles can be opened for write access to the named pipe.

CreateDisposition - Specifies the action to be taken if the named pipe does or does not already exist.

CreateDisposition Values:

FILE_CREATE - Indicates that if the named pipe already exists, then the operation should fail. If the named pipe does not already exist, then the first instance of the named pipe should be created.

FILE_OPEN - Indicates that if the named pipe already exists, then another instance of the named pipe should be created. If the named pipe does not already exist, then the operation should fail.

FILE_OPEN_IF - Indicates that if a named pipe already exists, then another instance of the named pipe should be created. If the named pipe does not already exist, then the first instance of the named pipe should be created.

CreateOptions − Specifies the options that should be used when creating the first instance or a subsequent instance of a named pipe.

CreateOptions Flags:

FILE_SYNCHRONOUS_IO_ALERT - Indicates that all operations on the named pipe are to be performed synchronously. Any wait that is performed on behalf of the caller is subject to premature termination by alerts.

FILE_SYNCHRONOUS_IO_NONALERT - Indicates that all operations on the named pipe are to be performed synchronously. Any wait that is performed on behalf of the caller is not subject to premature termination by alerts.

NamedPipeType - Specifies the type of the named pipe. This parameter is only meaningful when the first instance of a named pipe is created.

NamedPipeType Values:

FILE_PIPE_MESSAGE_TYPE - Indicates that the named pipe is a message pipe. Data written to the pipe is stored such that message boundaries are maintained. Message named pipes can be read in message mode or in byte stream mode.

FILE_PIPE_BYTE_STREAM_TYPE - Indicates that the named pipe is a byte stream pipe. Data written to the pipe is stored as a continuous stream of bytes. Byte stream pipes can only be read in byte stream mode.

ReadMode - Specifies the mode in which the named pipe is read.

ReadMode Values:

FILE_PIPE_MESSAGE_MODE - Indicates that data is read from the named pipe a message at a time. This value may not be specified unless the named pipe is a message pipe.

FILE_PIPE_BYTE_STREAM_MODE - Indicates that data is read from the named pipe as a continuous stream of bytes. This value may be specified regardless of the type of the named pipe.

CompletionMode - Specifies whether I/O operations are to be queued or completed immediately when conditions are such that the I/O operation cannot be completed without being deferred for subsequent processing, e.g., a read operation on a named pipe that contains no write data.

CompletionMode Values:

FILE_PIPE_QUEUE_OPERATION - Indicates that I/O operations are to be queued pending completion at a later time if they cannot be immediately completed when the I/O operation is issued.

FILE_PIPE_COMPLETE_OPERATION - Indicates that I/O operations are not to be queued if they cannot be completed immediately when the I/O operation is issued.

MaximumInstances - Specifies the maximum number of simultaneous instances of the named pipe. This parameter is only meaningful when the first instance of a named pipe is created.

InboundQuota - Specifies the pool quota that is reserved for writes to the inbound side of the named pipe.

OutboundQuota - Specifies the pool quota that is reserved for writes to the outbound side of the named pipe.

DefaultTimeout - Specifies an optional pointer to a timeout value that is to be used if a timeout value is not specified when waiting for an instance of a named pipe. This parameter is only meaningful when the first instance of a named pipe is created.

This service either creates the first instance of a specific named pipe and establishes its basic attributes or creates a new instance of an existing named pipe which inherits the attributes of the first instance of the named pipe. If creating a new instance of an existing named pipe the user must have *FILE_CREATE_PIPE_INSTANCE* access to the named pipe object.

If a new named pipe is being created, then the Access Control List (ACL) from the object attributes parameter defines the discretionary access control for the named pipe. If a new instance of an existing named pipe is created, then the ACL is ignored.

If a new named pipe is created, then the configuration of the named pipe is determined from the *FILE_SHARE_READ* and *FILE_SHARE_WRITE* flags of the share access parameter. If both flags are specified, then the named pipe is a full duplex pipe and can be read and written by clients. If either one or the other is specified, but not both, then the named pipe is a simplex pipe and can only be read (outbound) or written (inbound) by clients. If neither one is specified, then

STATUS_INVALID_PARAMETER is returned. If a new instance of an existing named pipe is created, then the share access parameter is ignored.

If a new named pipe is created, then the type of the named pipe, the maximum instances, and the default timeout value are taken from their corresponding parameters. If a new instance of an existing named pipe is created, then these parameters are ignored.

The create options, completion mode, and read mode are set to their specified values.

The actual pool quota that is reserved for each side of the named pipe is either the system default, the system minimum, the system maximum, or the specified quota rounded up to the next allocation boundary.

The name of the named pipe is taken from the object attributes parameter, which must be specified.

An instance of a named pipe is always deleted when the last handle to the instance of the named pipe is closed.

If *STATUS_SUCCESS* is returned as the service status, then a new instance of a named pipe was successfully created. The *Information* field of the I/O status block indicates if this is the first instance of the named pipe (*FILE_CREATED*) or a new instance of an existing named pipe (*FILE_OPENED*).

If *STATUS_INVALID_PARAMETER* is returned as the service status, then an invalid value was specified for one or more of the input parameters.

If *STATUS_INSTANCE_NOT_AVAILABLE* is returned as the service status, the named pipe already exists and creating another instance would cause the maximum number of instances to be exceeded.

5.2 Create File

The **NtCreateFile** function can be used to open a client end handle to an instance of a specified named pipe.

In order to use this function to open a named pipe, the named pipe must already exist and the *CreateDisposition* value must be specified as either *FILE_OPEN* or *FILE_OPEN_IF*.

When a named pipe is opened, a search is conducted for an available instance of the specified named pipe. If an instance of the named pipe is found that has a state of listening, then the state of the named pipe is set to connected, the read mode is set to byte stream, the completion mode is set to queued operation, and the open I/O request is completed. If one or more listen I/O requests are pending for the server end of the named pipe, then the listen I/O requests are completed with a status of *STATUS_SUCCESS*.

If a named pipe of specified name cannot be found, then *STATUS_OBJECT_NAME_NOT_FOUND* is returned as the service status.

If an instance of the named pipe cannot be found with a state of listening, then *STATUS_PIPE_NOT_AVAILABLE* is returned as the service status.

5.3 Open File

The **NtOpenFile** function can be used to open a client end handle to an instance of a specified named pipe.

When a named pipe is opened, a search is conducted for an available instance of the specified named pipe. If an instance of the named pipe is found that has a state of listening, then the state of the named pipe is set to connected, the read mode is set to byte stream, the completion mode is set to queued operation, and the open I/O request is completed. If one or more listen I/O requests are pending for the server end of the named pipe, then the listen I/O requests are completed with a status of *STATUS_SUCCESS.*

If a named pipe of specified name cannot be found, then *STATUS_OBJECT_NAME_NOT_FOUND* is returned as the service status.

If an instance of the named pipe cannot be found with a state of listening, then *STATUS_PIPE_NOT_AVAILABLE* is returned as the service status.

5.4 Read File

The **NtReadFile** function can be used to read data from a named pipe. Data is read according to the read mode of the specified named pipe and I/O operations are completed according to the completion mode of the specified named pipe.

The byte offset and key parameters of the **NtReadFile** function are ignored by the named pipe file system.

The specified named pipe must be in the connected or closing state in order to read information from the pipe.

If *STATUS_PENDING* is returned as the service status, then the read I/O operation is pending and its completion must be synchronized using the standard **NT OS/2** mechanisms. Any other service status indicates that the read I/O operation has already been completed. If a success status is returned, then the I/O status block contains the I/O completion information. Otherwise, the service status determines any error that may have occurred.

If the specified handle is not open to a named pipe that is in the connected or closing state, then *STATUS_INVALID_PIPE_STATE* is returned as the service status.

If the I/O status *STATUS_ACCESS_VIOLATION* is returned, then part of the read buffer became inaccessible after it was probed for write access and the I/O status block contains the number of bytes that were read.

If the I/O status *STATUS_END_OF_FILE* is returned, then there is no data in the pipe and the write end of the pipe has been closed.

The I/O status *STATUS_PIPE_EMPTY* is returned when there is no data in the pipe but the write end of the pipe is still opened and the pipe is opened for complete operations.

If the I/O status *STATUS_BUFFER_OVERFLOW* is returned, then the read I/O operation was completed successfully, but the size of the input buffer was not large enough to hold the entire input message. A full buffer of data is returned; additional data can be read from the message using the **NtReadFile** function. The I/O status block contains the number of bytes that were read.

If the I/O status *STATUS_SUCCESS* is returned, then the read I/O operation was completed successfully and the I/O status block contains the number of bytes that were read.

If an event is associated with the write end of the specified named pipe and any data is actually read from the pipe, then the event is set to the Signaled state. Writers can use this information to synchronize their access to the named pipe.

5.5 Write File

The **NtWriteFile** function can be used to write data to a named pipe. Data is written according to the type of the specified named pipe and I/O operations are completed according to the completion mode of the specified named pipe.

The byte offset and key parameters of the **NtWriteFile** function are ignored by the named pipe file system.

The specified named pipe must be in the connected state in order to write information to the pipe.

If *STATUS_PENDING* is returned as the service status, then the write I/O operation is pending and its completion must be synchronized using the standard **NT OS/2** mechanisms. Any other service status indicates that the write I/O operation has already been completed. If a success status is returned, then the I/O status block contains the I/O completion information. Otherwise, the service status determines any error that may have occurred.

If the specified handle is not open to a named pipe that is in the connected state, then *STATUS_INVALID_PIPE_STATE* is returned as the service status.

If the I/O status *STATUS_ACCESS_VIOLATION* is returned, then part of the write buffer became inaccessible after it was probed for read access and the I/O status block contains the number of bytes that were written.

If the I/O status *STATUS_SUCCESS* is returned, then the write I/O operation was completed successfully and the I/O status block contains the number of bytes that were written.

If an event is associated with the read end of the specified named pipe and any data is actually written to the pipe, then the event is set to the Signaled state. Readers can use this information to synchronize their access to the named pipe.

A zero length write to a message type pipe adds a logical EOF to the to the pipe.

5.6 Read Terminal File

This function is not supported by named pipes.

5.7 Query Directory Information

The **NtQueryDirectoryFile** function can be used to enumerate files within the root named pipe file system directory (i.e., "\Device\NamedPipe\"). All the standard **NT OS/2** information classes are supported. **NtOpenFile** is used to open the root named pipe directory. This function is not supported for remote named pipes.

5.8 Notify Change Directory

The **NtNotifyChangeDirectoryFile** function can be used to monitor modifications to the root named pipe file system directory. The standard **NT OS/2** capabilities are supported. This function is not supported for remote named pipes.

5.9 Query File Information

Information about a file can be obtained with the **NtQueryInformationFile** function. All information classes, with the exception of extended attribute information, are supported for named pipes with special interpretation of the returned data as appropriate. An additional information class is also provided to return information that is specific to named pipes.

Information is returned by the named pipe file system for named pipes and for the named pipe root directory. The following subsections describe the information that is returned for named pipe entries. The information returned for the root directory is identical to the information that is returned by other file systems and is described in the **NT OS/2** I/O System Specification.

5.9.1 Basic Information

Basic information about a named pipe includes the creation time, the time of the last access, the time of the last write, the time of the last change, and the attributes of the named pipe. The file attribute value for a named pipe is **FILE_ATTRIBUTE_NORMAL**. This function is only supported by local named pipes.

5.9.2 Standard Information

Standard information about a named pipe includes the allocation size, the end of file offset, the device type, the number of hard links, whether a delete is pending, and the directory indicator. This function is only supported by local named pipes.

The allocation size is the amount of pool quota charged to the creator of an instance of a named pipe. This is the sum of the quota charged for the inbound and outbound buffers. The end of file offset is the number of bytes that are available in the inbound buffer. The device type is **FILE_DEVICE_NAMED_PIPE**, the number of hard links is one, delete pending is TRUE, and the directory indicator is FALSE.

5.9.3 Internal Information

Internal information about a named pipe includes a named pipe file-system-specific identifier. This value is unique for each instance of a named pipe.

5.9.4 Extended Attribute Information

The extended attribute information size is always returned as zero by the named pipe file system. This function is only supported by local named pipes.

5.9.5 Access Information

Access information about a named pipe includes the granted access flags. This function is only supported by local named pipes.

5.9.6 Name Information

Name information about a named pipe includes the name of the named pipe. This function is only supported by local named pipes.

5.9.7 Position Information

Position information about a named pipe includes the current byte offset. The current byte offset is the number of bytes that are available in the input buffer. This function is only supported by local named pipes.

5.9.8 Mode Information

Mode information about a named pipe includes the I/O mode of the named pipe. This function is only supported by local named pipes.

5.9.9 Alignment Information

The alignment information class is not supported by the named pipe file system. This function is only supported by local named pipes.

5.9.10 All Information

The all information class includes information that can be returned by all file systems and is described above under each of the individual subsections. This function is only supported by local named pipes.

5.9.11 Pipe Information

Pipe information for both local and remote named pipes include the read and completion mode for the specified end of the named pipe. An access of FILE_READ_ATTRIBUTE is required to query the pipe information of a named pipe.

FilePipeQueryInformation - Data type is *FILE_PIPE_INFORMATION*.

typedef struct **_FILE_PIPE_INFORMATION** {
    ULONG *ReadMode*;
    ULONG *CompletionMode*;
} **FILE_PIPE_INFORMATION**;

FILE_PIPE_INFORMATION:

ReadMode - The mode in which the named pipe is being read (*FILE_PIPE_MESSAGE_MODE* or *FILE_PIPE_BYTE_STREAM_MODE*).

CompletionMode - The mode in which I/O operations are handled (*FILE_PIPE_QUEUE_OPERATION* or *FILE_PIPE_COMPLETE_OPERATION*).

5.9.12 Local Pipe Information

Information for a local named pipe includes the type of the pipe, the maximum number of instances of the named pipe that can be created, the current number of instances of the named pipe, the quota charged for the input buffer, the number of bytes of data available in the input buffer, the quota charged for the output buffer, the quota available for writing into the output buffer, the state of the named pipe, and the end of the named pipe. An access of FILE_READ_ATTRIBUTE is required to query the local pipe information of a named pipe. This function is only supported by local named pipes.

FilePipeQueryInformation - Data type is *FILE_PIPE_LOCAL_INFORMATION*.

typedef struct **_FILE_PIPE_LOCAL_INFORMATION** {
    ULONG *NamedPipeType*;
    ULONG *NamedPipeConfiguration*;
    ULONG *MaximumInstances*;
    ULONG *CurrentInstances*;
    ULONG *InboundQuota*;
    ULONG *ReadDataAvailable*;
    ULONG *OutboundQuota*;
    ULONG *WriteQuotaAvailable*;
    ULONG *NamedPipeState*;
    ULONG *NamedPipeEnd*;
} **FILE_PIPE_LOCAL_INFORMATION**;

FILE_PIPE_LOCAL_INFORMATION:

NamedPipeType - The type of the named pipe (*FILE_PIPE_MESSAGE_TYPE* or *FILE_PIPE_BYTE_STREAM_TYPE*).

NamedPipeConfiguration - The configuration of the named pipe (*FILE_PIPE_INBOUND*, *FILE_PIPE_OUTBOUND*, *FILE_PIPE_FULL_DUPLEX*).

MaximumInstances - The maximum number of simultaneous instances of the named pipe that are allowed.

CurrentInstances - The current number of instances of the named pipe. For a remote named pipe this field is set to **MAXULONG**.

InboundQuota - The amount of pool quota that is reserved for buffering writes to the inbound side of the named pipe. For a remote named pipe this field is set to **MAXULONG**.

ReadDataAvailable - The number of bytes of read data that are available in the input buffer. For a remote named pipe this field is set to **MAXULONG**.

OutboundQuota - The amount of pool quota that is reserved for buffering writes to the outbound side of the named pipe. For a remote named pipe this field is set to **MAXULONG**.

WriteQuotaAvailable - The number of bytes of pool quota that are available for writing data. For a remote named pipe this field is set to **MAXULONG**.

NamedPipeState - The current state of the named pipe (*FILE_PIPE_DISCONNECTED_STATE*, *FILE_PIPE_LISTENING_STATE*, *FILE_PIPE_CONNECTED_STATE*, or *FILE_PIPE_CLOSING_STATE*).

NamedPipeEnd - The end of the pipe that is referred to by the specified open file handle (*FILE_PIPE_CLIENT_END* or *FILE_PIPE_SERVER_END*).

5.9.13 Remote Pipe Information

Information for a remote named pipe includes the collect data time and the maximum collection count for the specified named pipe. An access of FILE_READ_ATTRIBUTE is required to query the pipe information of a named pipe. This function is only supported by remote named pipes.

FilePipeQueryInformation - Data type is *FILE_PIPE_REMOTE_INFORMATION*.

typedef struct **_FILE_PIPE_REMOTE_INFORMATION** {
    TIME *CollectDataTime*;
    ULONG *MaximumCollectionCount*;
} **FILE_PIPE_REMOTE_INFORMATION**;

FILE_PIPE_REMOTE_INFORMATION:

CollectDataTime - Specifies the amount of time that the workstation collects data to send to the remote named pipe before it sends it.

MaximumCollectionCount - Specifies the maximum number of bytes that the workstation stores before it sends data to the remote named pipe.

5.10 Set File Information

Information about a file can be changed with the **NtSetInformationFile** function. Most information classes are supported for local named pipes with the exception of link and position information.

Information can be set for named pipes. The following subsections describe the information that can be set for named pipes.

5.10.1 Basic Information

Basic information about a named pipe that can be set includes the creation time, the time of the last access, the time of the last write, the time of the last change, and the attributes of the named pipe. This function is only supported by local named pipes.

The associated times included in this class can be set to any appropriate value. The file attribute field can only be set to **FILE_ATTRIBUTE_NORMAL**.

5.10.2 Disposition Information

The disposition information class is not supported by named pipes.

Named pipes are always considered temporary and are deleted when the last handle is closed (i.e., when the last instance of a named pipe is closed and deleted the named pipe, itself, is also deleted.

5.10.3 Link Information

This information class is not supported by named pipes.

5.10.4 Position Information

This information class is not supported by named pipes.

5.10.5 Mode Information

Mode information about a named pipe that can be set includes the I/O mode of the named pipe.

5.10.6 Pipe Information

Pipe information about a named pipe that can be set includes the read mode and completion mode of the named pipe. No special access is required to set the pipe information.

FilePipeSetInformation - Data type is *FILE_PIPE_INFORMATION*.

```
typedef struct _FILE_PIPE_INFORMATION {
    ULONG ReadMode;
    ULONG CompletionMode;
} FILE_PIPE_INFORMATION;
```

FILE_PIPE_INFORMATION:

ReadMode - The mode in which the named pipe is to be read (*FILE_PIPE_MESSAGE_MODE* or *FILE_PIPE_BYTE_STREAM_MODE*).

CompletionMode - The mode in which I/O operations are to be handled (*FILE_PIPE_QUEUE_OPERATION* or *FILE_PIPE_COMPLETE_OPERATION*).

If the type of the specified named pipe is a byte stream pipe and the new read mode is message mode, then *STATUS_INVALID_PARAMETER* is returned as the service status.

If the new completion mode for the specified named pipe is complete operations, the current completion mode is queue operations, and one or more I/O operations are currently queued to the specified end of the named pipe, then *STATUS_PIPE_BUSY* is returned as the service status and no pipe information is changed.

If the new read mode and the new completion mode are compatible with the current state of the specified named pipe, then the set information I/O request is completed with a status of *STATUS_SUCCESS* and *STATUS_SUCCESS* is returned as the service status.

5.10.7 Remote Pipe Information

Information about a remote named pipe that can be set includes the collect data time and the maximum collection count. No special access is required to set the pipe information.

FilePipeSetInformation - Data type is *FILE_PIPE_REMOTEINFORMATION*.

typedef struct **_FILE_PIPE_REMOTE_INFORMATION** {
    TIME *CollectDataTime*;
    ULONG *MaximumCollectionCount*;
} **FILE_PIPE_REMOTE_INFORMATION**;

FILE_PIPE_REMOTE_INFORMATION:

CollectDataTime - Sets the amount of time that the workstation can collect before sending it to the remote named pipe.

MaximumCollectionCount - Sets the maximum number of bytes that the workstation stores before sending data to the remote named pipe.

5.11 Query Extended Attributes

This function is not supported by named pipes.

5.12 Set Extended Attributes

This function is not supported by named pipes.

5.13 Lock Byte Range

This function is not supported by named pipes.

5.14 Unlock Byte Range

This function is not supported by named pipes.

5.15 Query Volume Information

This function is not supported by named pipes.

5.16 Set Volume Information

This function is not supported by named pipes.

5.17 File Control Operations

The following subsections describe file control operations that can be performed using a handle that is open to an instance of a named pipe. Certain functions can only be executed using a handle that is open to the server end of a named pipe. These functions are not legal for a handle that is open to the client end of a named pipe. The wait for named pipe instance function and the query event information function both require a handle that is open to the named pipe file system itself.

5.17.1 External File Control Operations

External file control operations can be executed by all users of the **NT OS/2** named pipe facilities and do not require any special privileges.

5.17.1.1 Assign Event

The assign event file control operation associates or disassociates an event object with either the client or server end of a named pipe. This function is only supported by local named pipes.

The control code for this operation is *FSCTL_PIPE_ASSIGN_EVENT*. The input buffer parameter specifies the event handle and key value that are to be associated with the respective end of the named pipe. The input buffer has the following format:

typedef struct **_FILE_PIPE_ASSIGN_EVENT_BUFFER** {
    HANDLE *EventHandle*;
    ULONG *KeyValue*;
    } **FILE_PIPE_ASSIGN_EVENT_BUFFER**;

FILE_PIPE_ASSIGN_EVENT_BUFFER:

EventHandle - A handle to an event object that is to be associated with the respective end of the named pipe, or null if the currently associated event object is to be disassociated.

KeyValue - The key value that is to be associated with the respective end of the named pipe. If the event handle is null, then this parameter is ignored.

If the event handle is null, then any event object that is currently associated with the respective end of the named pipe is disassociated and the key value is ignored.

If the event handle is not null, then **WRITE** access to the event is required. Any previously associated event object is disassociated and the specified event and key value are associated with the respective end of the named pipe.

This operation is always completed immediately and never causes an I/O operation to be queued.

Assigning an event object to either the client or server end of a named pipe provides additional synchronization capabilities when I/O operations are completed immediately rather than being queued.

Once an event object is assigned, the event will be set to the Signaled state every time information is read from, or written to, the opposite end of the named pipe, or the opposite end of the named pipe is closed. The event object associated with the client end of the named pipe is also set to the Signaled state when a disconnect operation is performed on the server end of the pipe.

5.17.1.2 Disconnect

The disconnect file control operation disconnects an instance of a named pipe from a client and causes the named pipe to enter the disconnected state. Disconnecting a named pipe causes all data in the pipe to be discarded and no further access to the named pipe is allowed until a listen operation is performed. The function is only valid from the server end of a named pipe.

The control code for this operation is *FSCTL_PIPE_DISCONNECT*. The input and output parameter buffers are not used.

If the specified handle is not open to the server end of a named pipe, then *STATUS_ILLEGAL_FUNCTION* is returned as the service status.

If the named pipe associated with the specified handle is in the disconnected state, then *STATUS_PIPE_DISCONNECTED* is returned as the service status.

If the named pipe associated with the specified handle is in the listening state, then the state of the named pipe is set to disconnected. If one or more listen I/O requests are waiting for a companion client open request, then the listen I/O requests are completed with a status of *STATUS_PIPE_DISCONNECTED*. The disconnect I/O request is completed with a status of *STATUS_SUCCESS* and *STATUS_SUCCESS* is returned as the service status.

If the named pipe associated with the specified handle is in the connected state, then the state of the named pipe is set to disconnected, all data in the input and output buffers is discarded, and outstanding client and server read and write I/O requests are completed with a status of *STATUS_PIPE_DISCONNECTED*. If an event object is associated with the client end of the named pipe, then the event is set to the Signaled state. The disconnect I/O request is completed with a status of *STATUS_SUCCESS* and *STATUS_SUCCESS* is returned as the service status.

If the named pipe associated with the specified handle is in the closing state, then the state of the named pipe is set to disconnected, all data in the input buffer is discarded, and outstanding server read I/O requests are completed with a status of *STATUS_PIPE_DISCONNECTED*. The disconnect I/O request is completed with a status of *STATUS_SUCCESS* and *STATUS_SUCCESS* is returned as the service status.

5.17.1.3 Listen

The listen file control operation is used to transition a named pipe from a disconnected state to a listening state. When a named pipe is in the listening state, client open requests can be satisfied and cause the named pipe to transition to the connected state. This function is only supported by local named pipes.

The control code for this operation is *FSCTL_PIPE_LISTEN*. The input and output parameter buffers are not used.

If the specified handle is not open to the server end of a named pipe, then *STATUS_ILLEGAL_FUNCTION* is returned as the service status.

If the named pipe associated with the specified handle is in the closing state, then *STATUS_PIPE_CLOSING* is returned as the service status.

If the named pipe associated with the specified handle is in the connected state, then *STATUS_PIPE_CONNECTED* is returned as the service status.

If the named pipe associated with the specified handle is in the listening state and the completion mode associated with the server end handle is queue operations, then the listen I/O request is queued awaiting a companion client open request and *STATUS_PENDING* is returned as the service status. Otherwise (the completion mode is complete operations), *STATUS_PIPE_LISTENING* is returned as the service status.

If the named pipe associated with the specified handle is in the disconnected state, then the state of the pipe is set to listening and any outstanding wait for named pipe I/O requests are completed with a status of *STATUS_SUCCESS*.

If the completion mode associated with the server end handle is complete operations, then the listen I/O request is completed with an I/O status of *STATUS_PIPE_LISTENING* and *STATUS_SUCCESS* is returned as the service status.

If the completion mode associated with the server end handle is queue operations, then the listen I/O request is queued awaiting a companion client open request and *STATUS_PENDING* is returned as the service status. When a client open is performed, the listen I/O request is completed with an I/O status of *STATUS_PIPE_CONNECTED*.

5.17.1.4 Peek

The peek file control operation reads data from a named pipe in either byte stream or message mode, but does not actually remove the data from the pipe.

The control code for this operation is *FSCTL_PIPE_PEEK*. The output buffer parameter specifies the read buffer for the peek operation. The output buffer has the following format:

typedef struct **_FILE_PIPE_PEEK_BUFFER** {
ULONG *NamedPipeState*;
ULONG *ReadDataAvailable*;
ULONG *NumberOfMessages*;
ULONG *MessageLength*;
CHAR *Data*[];
} **FILE_PIPE_PEEK_BUFFER**;

FILE_PIPE_PEEK_BUFFER:

NamedPipeState - The current state of the named pipe (*FILE_PIPE_DISCONNECTED_STATE, FILE_PIPE_LISTENING_STATE, FILE_PIPE_CONNECTED_STATE*, or *FILE_PIPE_CLOSING_STATE*).

ReadDataAvailable - The number of bytes of read data that are available in the input buffer.

NumberOfMessages - The number of messages that are currently in the named pipe. If the named pipe is a message pipe, then this field contains the number of messages. Otherwise, this field contains zero.

MessageLength - The number of bytes that are contained in the first message in the named pipe. If the named pipe is a message type pipe, then this field contains the size of the first message. Otherwise, this field contains zero.

Data - A buffer that receives data read from the named pipe. The number of bytes of data that were read from the named pipe can be calculated from the I/O status block.

The specified named pipe must be in the connected or closing state in order to read information from the pipe.

This function is nearly identical to the **NtReadFile** function for a named pipe; however, no data is actually removed from the pipe and the operation is always completed immediately, i.e., it never causes an I/O operation to be queued.

If the specified handle is not open to a named pipe that is in the connected or closing state, then *STATUS_INVALID_PIPE_STATE* is returned as the service status.

If the I/O status *STATUS_ACCESS_VIOLATION* is returned, then part of the output buffer became inaccessible after it was probed for write access and the I/O status block contains the number of bytes that were read.

If the I/O status *STATUS_END_OF_FILE* is returned, then there is no data in the pipe and the write end of the pipe has been closed.

If the I/O status *STATUS_BUFFER_OVERFLOW* is returned, then the peek I/O operation was completed successfully, but the size of the output buffer was not large enough to hold the entire input

message. A full buffer of data is returned; the actual message size can be determined from information placed in the output buffer. The I/O status block contains the number of bytes that were read including the named pipe information.

If the I/O status *STATUS_SUCCESS* is returned, then the peek I/O operation was completed successfully and the I/O status block contains the number of bytes that were read including the named pipe information.

5.17.1.5 Query Event Information

The query event information file control operation returns information about each named pipe that a specified event object is associated with in the current process. It does not return information about named pipes that are associated with the specified event object in other processes. This function can only be executed using a handle that is open to the named pipe file system itself. This function is only supported by local named pipes.

The control code for this operation is *FSCTL_PIPE_QUERY_EVENT*. The input buffer specifies the handle for the event object that is to be queried. The output buffer parameter specifies the information buffer for the query operation. Each entry returned in the output buffer has the following format:

typedef struct **_FILE_PIPE_EVENT_BUFFER** {
ULONG *NamedPipeState*;
ULONG *EntryType*;
ULONG *ByteCount*;
ULONG *KeyValue*;
ULONG *NumberRequests*;
} **FILE_PIPE_EVENT_BUFFER**;

FILE_PIPE_EVENT_BUFFER:

NamedPipeState - The current state of the named pipe (*FILE_PIPE_DISCONNECTED_STATE, FILE_PIPE_LISTENING_STATE, FILE_PIPE_CONNECTED_STATE*, or *FILE_PIPE_CLOSING_STATE*).

EntryType - The type of entry (*FILE_PIPE_READ_DATA* or *FILE_PIPE_WRITE_SPACE*).

ByteCount - The number of bytes of read data that are available (entry type is *FILE_PIPE_READ_DATA*) or the number of bytes of available write space (entry type is *FILE_PIPE_WRITE_SPACE*).

KeyValue - The key value that is associated with the named pipe.

NumberRequests - The number of read I/O requests that are queued (entry type is *FILE_PIPE_WRITE_SPACE*) or the number of write I/O requests that are queued (entry type is *FILE_PIPE_READ_DATA*) to the opposite end of the named pipe.

This operation is always completed immediately and never causes an I/O operation to be queued.

If a named pipe that is associated with the specified event has both read data available and write space available, then two entries are returned in the output buffer.

If the specified handle is not an event object, then *STATUS_INVALID_PARAMETER* is returned as the service status.

If the I/O status *STATUS_ACCESS_VIOLATION* is returned, then part of the information buffer became inaccessible after it was probed for write access and the I/O status block contains the number of bytes of information that were returned.

If the I/O status *STATUS_SUCCESS* is returned, then the query event I/O operation was completed successfully and the I/O status block contains the number of bytes of information that were returned.

5.17.1.6 Transceive

The transceive file control operation performs a write operation followed by a read operation on a named pipe such that no other operation can occur between the write and read operations on the corresponding end of the pipe.

The control code for this operation is *FSCTL_PIPE_TRANSCEIVE*. The output buffer parameter specifies the read buffer and the input buffer parameter specifies the data to be written.

The specified named pipe must be in the connected state in order to perform a transceive operation on the pipe. The named pipe must also be a message pipe, and the read mode of the named pipe must be message mode. The completion mode is ignored for the transceive operation and operations are always queued.

If *STATUS_PENDING* is returned as the service status, then the transceive I/O operation is pending and its completion must be synchronized using standard **NT OS/2** mechanisms. Any other service status indicates that the transceive I/O operation has already been completed. If a success status is returned, then the I/O status block contains the I/O completion information. Otherwise, the service status determines any error that may have occurred.

If the specified handle is not open to a named pipe that is in the connected state, then *STATUS_INVALID_PIPE_STATE* is returned as the service status.

If the read mode associated with the specified handle is not message mode, then *STATUS_INVALID_READ_MODE* is returned as the service status.

If a read I/O operation is already pending for the inbound side of the specified named pipe, or there is currently available data in the inbound side of the named pipe, then *STATUS_PIPE_BUSY* is returned as the service status.

If the I/O status *STATUS_ACCESS_VIOLATION* is returned, then part of the read buffer or the write buffer became inaccessible after it was probed for write access (read buffer) or read access (write buffer) and the I/O status block contains the number of bytes that were read.

If the I/O status *STATUS_BUFFER_OVERFLOW* is returned, then the transceive I/O operation was completed successfully, but the size of the output buffer was not large enough to hold the entire input message. A full buffer of data is returned; additional data can be read from the message using the **NtReadFile** function. The I/O status block contains the number of bytes that were read.

If the I/O status *STATUS_SUCCESS* is returned, then the transceive I/O operation was completed successfully and the I/O status block contains the number of bytes that were read.

If an event is associated with the opposite end of the specified named pipe, then the event is set to the Signaled state when the write part of the transceive operation is completed and when the read part of the transceive operation is completed. Readers and writers can use this information to synchronize their access to the named pipe.

5.17.1.7 Wait For Named Pipe

The wait for named pipe file control operation waits for an instance of a named pipe with a specified name to attain a state of listening. This function can only be executed using a handle that is open to the named pipe file system root directory (i.e., "\Device\NamedPipe\") or redirector (i.e., "\Device\LanmanRedirector").

The control code for this operation is *FSCTL_PIPE_WAIT*. The input buffer parameter specifies the device relative name of the named pipe, and an optional timeout value. The input buffer has the following format:

typedef struct **_FILE_PIPE_WAIT_FOR_BUFFER** {
    TIME *Timeout*;
    ULONG *NameLength*;
    BOOLEAN *TimeoutSpecified*;
    CHAR *Name[]*
    } **FILE_PIPE_WAIT_FOR_BUFFER**;

FILE_PIPE_WAIT_FOR_BUFFER:

Timeout - Supplies a new timeout value is use other than the default timeout for the named pipe. This value is only read if TimeoutSpecified is TRUE. A minimum large integer value (i.e., 0x8000000000000000) means to wait indefinitely.

NameLength - Supplies the length of the name of the named pipe found in this buffer.

TimeoutSpecified - Indicates if an overriding timeout value has been specified.

Name - Supplies the name of the named pipe. The name does not include the "\Device\NamedPipe\" or "\Device\LanmanRedirector\" prefix.

If an instance of a named pipe with the specified name is currently in the listening state, then the wait for named pipe I/O function is completed with a status of *STATUS_SUCCESS* and *STATUS_SUCCESS* is returned as the service status. Otherwise, the wait for named pipe I/O request is placed in the wait queue of the specified named pipe and *STATUS_PENDING* is returned as the service status.

If an instance of the specified named pipe does not attain a listening state within the specified timeout period (either the optional one supplied in this function or the default timeout period specified when the original instance of the named pipe was created). then the wait for named pipe I/O request is completed with a status of *STATUS_PIPE_WAIT_TIMEOUT*.

5.17.1.8 Impersonate

The impersonate file control operation allows the server end of the pipe to impersonate the client end. Whenever this function is called the named pipe file system changes the caller's thread to start impersonating the context of the last message read from the pipe. Only the server end of the pipe is allowed to invoke this function. This function is only supported by local named pipes.

The control code for this operation is *FSCTL_PIPE_IMPERSONATE*. The output and input parameter buffers are not used.

If the specified handle is not open to the server end of a named pipe, then *STATUS_ILLEGAL_FUNCTION* is returned as the service status.

If the named pipe associated with the specified handle is in the disconnected state, then *STATUS_PIPE_DISCONNTECTED* is returned as the service status.

If a read operation has never been completed to the server end of the named pipe, then *STATUS_CANNOT_IMPERSONATE* is returned as the service status.

If the impersonation is successful then the I/O function is completed with a status of *STATUS_SUCCESS* and *STATUS_SUCCESS* is returned as the service status.

5.17.2 Internal File Control Operations

Internal file control operations can only be executed by components that execute in kernel mode and directly build and submit I/O requests to the named pipe file system. These functions are only supported by local named pipes.

5.17.2.1 Internal Read

The internal read file control operation provides the capability to perform a read operation directly into a system buffer. No quota is charged nor are any buffers allocated by the named pipe file system.

The control code for this operation is *FSCTL_PIPE_INTERNAL_READ*. The output buffer parameter specifies the system buffer into which information is to be read

5.17.2.2 Internal Write

The internal write file control operation provides the capability to perform a write operation directly from a system buffer. No quota is charged nor are any buffers allocated by the named pipe file system.

The control code for this operation is *FSCTL_PIPE_INTERNAL_WRITE*. The input buffer parameter specifies the system buffer from which information is to be written.

5.17.2.3 Internal Transceive

The internal transceive control operation provides the capability to perform a transceive operation directly into a system buffer. No quota is charged nor are any buffers allocated by the named pipe file system.

The control code for this operation is *FSCTL_PIPE_INTERNAL_TRANSCIEVE*. The input buffer parameter specifies the buffer from which information is to be written, while the output buffer parameter specifies the system buffer into which information is to be read.

5.18 Flush Buffers

The **NtFlushBuffersFile** function can be used to wait until all currently buffered write data is read from the opposite end of the specified named pipe.

5.19 Set New File Size

This function is not supported by named pipes.

5.20 Cancel I/O Operation

The **NtCancelIoFile** function can be used to cancel all I/O operations that were issued by the subject thread for the specified named pipe. Both read and write operations initiated by the subject thread are canceled.

5.21 Device Control Operations

No device control operations are supported by the named pipe file system.

5.22 Close Handle

The **NtClose** function can be used to close a handle to the specified named pipe.

If the specified handle is the last handle that is open to the corresponding end of the specified named pipe, then the state of the named pipe is set to closing. Read and write operations that are pending for the inbound side of the named pipe are completed with an I/O status of *STATUS_PIPE_CLOSED*.

Write operations that are pending for the outbound side of the named pipe are allowed to complete and cause the close operation to remain pending until the opposite end of the named pipe is closed, disconnected, or the information is read from the pipe.

If an event is associated with the opposite end of the specified named pipe, then the event is set to the Signaled state. Readers and writers can use this information to synchronize their access to the named pipe.

6. OS/2 API Emulation

The following subsections discuss the emulation of the OS/2 named pipe facilities using the capabilities provided by **NT OS/2**. Only those OS/2 functions which require special handling with respect to named pipes are included.

6.1 DosCallNmPipe

This OS/2 API combines the function of an open, write, read, and a close of a named pipe.

This service can be emulated with the **NtOpen**, **NtFsControlFile** (*FSCTL_PIPE_TRANSCEIVE*), and **NtClose** services. There is no **NT OS/2** facility that will perform this function in a single operation.

6.2 DosConnectNmPipe

This OS/2 API causes an instance of a named pipe that is in the disconnected state to transition to the listening state and continues the execution of any clients that are waiting for an available instance of the specified named pipe. This function can only be executed using a handle that is associated with the server end of a named pipe.

This API can be emulated with the **NtFsControlFile** service by specifying a function code of *FSCTL_PIPE_LISTEN*. The OS/2 subsystem or client DLL issues the listen I/O request. If the completion mode associated with the specified named pipe handle is queue operations and the request cannot be immediately satisfied, then *STATUS_PENDING* is returned. For this case, the OS/2 subsystem or client DLL must wait for the I/O operation to complete.

6.3 DosDisconnectNmPipe

This OS/2 API causes an instance of a named pipe to enter the disconnected state. All data in the input and output buffers of the pipe are discarded and any outstanding read or write I/O requests are completed with an error status. This function can only be executed using a handle that is associated with the server end of a named pipe.

This API can be emulated with the **NtFsControlFile** service by specifying a function code of *FSCTL_PIPE_DISCONNECT*. The OS/2 subsystem or client DLL issues the disconnect I/O request.

6.4 DosMakeNmPipe

This OS/2 API creates an instance of a named pipe and opens a server side handle to the newly created instance. If the newly created instance is the first instance of the named pipe, then the attributes of the named pipe are also defined.

This API can be emulated with the **NtCreateNamedPipeFile** service.

The OS/2 inheritance bit of the open mode is the same as the **NT OS/2** handle attributes field of the object attributes parameter.

The OS/2 write-behind bit of the open mode is the opposite of the **NT OS/2** *FILE_WRITE_THROUGH* flag of the create options parameter. Therefore, a particular OS/2-compatible behavior can be specified with the **NT OS/2** parameter.

The OS/2 access bits of the open mode are the same as the **NT OS/2** desired access parameter.

The **NT OS/2** share access flags are used to determine the configuration of the named pipe (i.e., full duplex or simplex).

The OS/2 wait bit of the pipe mode is the same as the **NT OS/2** completion mode parameter.

The OS/2 read bit of the pipe mode is the same as the **NT OS/2** read mode parameter.

The OS/2 pipe type bit of the pipe mode is the same as the **NT OS/2** pipe type parameter.

The OS/2 maximum instances field of the pipe mode is the same as the **NT OS/2** maximum instances parameter.

The OS/2 outbound buffer size is the same as the **NT OS/2** outbound quota parameter.

The OS/2 inbound buffer size is the same as the **NT OS/2** inbound quota parameter.

The OS/2 default timeout is the same as the **NT OS/2** default timeout parameter.

6.5 DosPeekNmPipe

This OS/2 API allows information to be read from a named pipe without actually removing the data from the pipe.

This API can be emulated with **NtFsControlFile** service by specifying a function code of *FSCTL_PIPE_PEEK*. The OS/2 subsystem or client DLL issues the peek I/O request. The request is completed immediately and the information returned in the output buffer and I/O status block can be used to generate the output values required by the OS/2 API.

6.6 DosQNmPHandState

This OS/2 API returns information about the instance of a named pipe that is open to the specified handle.

This API can be emulated with the **NtQueryInformationFile** service by specifying the *FilePipeQueryInformation* information class.

6.7 DosQNmPipeInfo

This OS/2 API returns information about the instance of a named pipe that is open to the specified handle.

This API can be emulated with the **NtQueryInformationFile** service by specifying the *FilePipeQueryInformation* and *FileNameInformation* information classes.

6.8 DosQNmPipeSemState

This OS/2 API returns information about all named pipes that are associated with a specified semaphore handle.

This API can be emulated with **NtFsControlFile** service by specifying a function code of *FSCTL_PIPE_QUERY_EVENT*. The OS/2 subsystem or client DLL issues the query event I/O request. The request is completed immediately and the information returned in the output buffer and I/O status block can be used to generate the output values required by the OS/2 API.

6.9 DosRawReadNmPipe

This OS/2 API provides the capability to read all the available data, including message headers, from a named pipe.

This is an undocumented function in OS/2 and will not be implemented as a user-visible function by the OS/2 subsystem.

There seems to be no real use for this function.

6.10 DosRawWriteNmPipe

This OS/2 API provides the capability to write data, including message headers, to a named pipe.

This is an undocumented function in OS/2 and will not be implemented as a user-visible function by the OS/2 subsystem.

The only known user-level need for this function is to enable the writing of a zero length message to a message pipe. This capability will be provided in a different manner by the **NT OS/2** name pipe file system.

6.11 DosSetNmPHandState

This OS/2 API sets information about the instance of a named pipe that is open to the specified handle.

This API can be emulated with the **NtSetInformationFile** service by specifying the *FilePipeSetInformation* information class.

6.12 DosSetNmPipeSem

This API associates a semaphore and key value with a named pipe.

This API can be emulated with **NtFsControlFile** service by specifying a function code of *FSCTL_PIPE_ASSIGN_EVENT*.

6.13 DosTransactNmPipe

This OS/2 API combines the function of a write operation and a read operation on a named pipe. The transact operation is performed on the named pipe such that no other operation can occur between the write and read operations.

This API can be emulated with the **NtFsControlFile** service by specifying a function code of *FSCTL_PIPE_TRANSCEIVE* and then waiting for the I/O request to complete.

6.14 DosWaitNmPipe

This OS/2 API provides the ability for a client to wait until an instance of a named pipe with a specified name attains a state of listening.

This API can be emulated with the **NtFsControlFile** service by specifying a function code of *FSCTL_PIPE_WAIT* and then waiting for the I/O request to complete.

The I/O request will automatically be completed if the default timeout interval that was specified when the original instance of the named pipe was created is exceeded. If a timeout value is specified by the user, then the overriding timeout period should be used in the FSCTL pipe wait call.

Revision History:

Original Draft, February 16, 1990

Revision 1.1, March 8, 1990

1. Incorporate technical and editorial changes from internal review.

Revision 1.2, August 14, 1990

1. Removed directory hierarchy.
2. Removed raw mode read and write.
3. Added optionally timeout parameter to wait for named pipe.
4. Removed all references to EAs and symbolic links.
5. Minor editoral changes.

Revision 1.3, September 27, 1990

1. Removed owner information query/set operation.
2. Changed unbuffered read/write to internal read/write.
3. Added internal transceive operation.
4. Minor editoral changes.

Revision 1.4, October 17, 1990

1. Added impersonation.

Revision 1.5, January 23, 1991

1. Clarify that NtCreateNamedPipeFile and directory query are for local pipes only.
2. In query Pipe information state which fields remote pipes returns as **MAXULONG**.
3. Remove FILE_WRITE_THROUGH option in NtCreateNamedPipeFile.
4. Change wait for named pipe to take a handle to the root directory and not the file system itself.
5. Add remote named pipes.