UNIX NETWORK PROGRAMMING

Cork Regional Technical College (RTC)
B.SC. in Computer Applications, stage IV.

Data Communications, Assignment I

Lecturer : Mr. Jim O'Dwyer
Author : Stefan Krautwurst, DCOM4
Date : 1994/11/20

Conventions Used

Throughout this report, the word ‘STREAMS‘ will refer to the mechanism and the word ‘stream‘ (pl. ‘streams‘) will refer to the path between a user application and a driver.

Contents

1) Streams

1.1 Introduction: What is STREAMS?

STREAMS is a standardized mechanism for developing software capable for use in computer networks. It has been developed by Dennis Ritchie and was implemented for the first time in UNIX System V Release 3.

The STREAMS mechanism enables communications between user processes. It is a flexible facility providing a set of tools for development of UNIX communication programs and services.
The associated mechanism is simple and open-ended, consisting of a set of system calls, kernel resources, and kernel routines.

A stream is a full-duplex processing and data transfer path between a process in user space and a STREAMS driver in kernel space. This driver itself reacts to system calls of the user and controls the communication hardware. Thereby, the user only accesses the driver in between.

In the kernel, a stream consists of a stream head, a driver, and zero or more modules between both of them. The stream head is the end of the stream nearest to the user process. All system calls on a stream, made by a user process, are processed by the stream head.

1.2 Benefits of streams

Benefits of streams lie in their portability, flexibility and manipulative abilities on modules. As a consequence, it is easy to create, modify and integrate modules, serving as an interface to other processes. It also enables the integration of services directly accessing streams into already existing applications.

Because of this, streams support the dynamic linkage of layered network models. User level programs can be separated from underlying layers respectively protocols, which raises the transparency of the whole network architecture.

1.3 STREAMS driver

This driver may be a device driver that provides the services of an external I/O device. It can also be a software driver,
commonly referred to as a pseudo-device driver.
The driver typically handles data transfer between the kernel and the device. Normally there is done no processing of data
apart from the conversion between data structures used by STREAMS and data stuctures that the device understands.
Drivers are accessed via nodes in the file system and may be opened just like any other device.

1.4 STREAMS modules

A STREAMS module is a defined set of kernel-level routines and data structures used to process data, status, and control information. Data processing may involve changing the way the data are represented, adding/deleting header and trailer information to data, and/or packetizing/depacketizing data. Each module is self-contained and functionally isolated from any other component in the stream except its two neighbouring components. The module communicates with its neighbours by passing messages. The module is not a required component in STREAMS, whereas the driver is. Only in a STREAMS-based pipe a stream head is sufficient; there is no need for the driver.

1.5 Pipes

Pipes are also STREAMS-based. The stream system call PIPE()establishes a full-duplex, bidirectional data transfer path between the kernel and one or more user processes. This call creates two streams with each stream header representing one end of the pipe.

1.6 Messages

As already pointed out, modules are communicating by passing messages, which simply consist of a set of data structures.
These messages contain the data to be sent from the driver to the head. The processing takes place at the head and the data is sent furtheron to the buffers.
Data can also be sent the other way round (from the head to the driver).

A message consists of a linked list of message blocks. Each of these blocks is made of three parts: a header, a data block, and a data buffer.

Since a message may be a control or data message, there is a type indicator in the message header, providing the identifi-cation of the type.
Control messages may result from IOCTL system calls or from special conditions such as a terminal hangup. Data messages may result from WRITE system calls or when data arrives from a device.
It is also possible to change the priority of a device to a higher or lower value.

2) The programming interface to streams

2.1 Sockets

2.1.1 What is a socket?

Sockets have been implemented in UNIX in order to simplify interprocess communication and networking protocols. They function as communication endpoints of a two-way byte stream. A socket is used exactly in the same way as a file descriptor. Nevertheless there is a difference: File descriptors have to be inherited from a parent process, whereas sockets can be created anywhere.

A list of the socket system calls is to be found in the appendix, section 3.3.

2.1.2 Use of sockets for interprocess communication

2.1.2.1 Creating a socket

To begin with, we need to create a socket. Therefore we have to think about the way we wish to transfer the data. Do we want to pass them to another process on the same machine (the socalled *UNIX-DOMAIN*), or shall they even be transferred to another machine (*INET-DOMAIN*)?

The first method establishs a connection similar to a pipe. Data is written via the first socket to a buffer and can be read via a second socket.
The *INET-DOMAIN*, on the other hand, provides protocols (UDP, TCP) for the transmission of the data in packets through the network.

A socket is created with the system call *socket(domain, type, protocol)*.
In order to create a datagram socket For local use, *domain* should be *AF_UNIX*.

example: *s=socket(AF_UNIX,SOCK_DGRAM,0)*.

For creating a stream socket within the *INET-DOMAIN* ,*AF_INET* should be used. This provides a creation of the socket with the TCP protocol including support for the communication.

example: *s=socket(AF_INET,SOCK_STREAM,0)*

A list of the possible *type*s of the system call socket are to be found in the appendix section 3.4.

The following errors may occur:

ENOBUFS	lack of memory
EPROTONOSUPPORT	unknown protocol
EPROTOTYPE	no supporting protocol

Note: There exists another system call for creating sockets,
namely *socketpair* with a fourth argument, *sarray*, for creating a pair of sockets at once. The two socket descriptors are placed in the two elements of *sarray*. This system call is useful for creating a pipe with two endpoints of communication.

2.1.2.2 The system call*bind*

Since a socket is created without any association to an address, there is the *bind* system call with the following form: *bind (socket, localaddr, addrlen)*, whereas

socket = descriptor (number) of the socket to be bound

localaddr = local address to bind the socket to

addrlen = length of the address in bytes

2.1.2.3 The system call*connect*

The system call *connect(socket, destaddr, addrlen)* provides the connection (binding) to a foreign address. The meaning of the three variables is again:

socket = descriptor (number) of the socket to be bound

destaddr = destination address to bind the socket to

addrlen = length of the destination address in bytes

Now the transfer of data can be started!

2.1.2.4 System calls for sending data

The following system calls can be used for sending data:

send enables sending a message *out_of_bands* on sockets

write writing a message

writev like *write*, but making it possible to write the message without copying it into continuous bytes of memory

sendto \ / both calls allow to send a message through an
sendmsg / \ unconnected socket, because they have to be
given the destination address as an additional
argument

The first three calls only work after a *connect*, because they do not enable to specify the destination address.

2.1.2.5 System calls for receiving data

The five corresponding system calls for receiving data are:
read, *readv*, *recv*, *recvfrom*, and *recvmsg*. The first three calls again only work for *connect*ed sockets.

2.1.2.6 System calls *close* and *discard*

Simply close a socket with the system call *close(socket)*, whereas *socket* =descriptor (number) of the socket to be bound.
Although this does not automatically stop the transfer of data, the data could not be delivered anymore and thus will be dis-carded after a while.

The system call *shutdown(socket,how)* may be performed before a close. *socket* again determines the descriptor, *how* determines if data shall stop being received, sent, or both.

2.2 The Transport Level Interface (TLI)

2.2.1 Overview

This interface is situated on top of the transport layer, as one may already deduce from its name. The transport layer provides the basis for reliable end to end data transfer and hides the underlying layers and protocols from the user.

The TLI defines a set of services similar to the layers of other protocol groups like ISO or SNA enabling compatible data transmission across these nets. It is implemented as a user library using streams input/output mechanism. Thus, many services available under streams can also be used here.

2.2.2 Types of messages between user and provider

Between the transport layer and the next (upper) session layer, messages are passing between entities. The user (upper layer entity) and the service provider (lower entity) pass 4 types of messages between each other:

1. request: the user requests the provider to do a certain task
2. confirmation: the provider sends the results to the user
3. indication: the provider informs the user about the occurrence of an event
4. response: the user responds to the provider's indication

2.2.3 Two service modes

2.2.3.1 Overview

The TLI provides two service modes:
The *connection mode* and the *connectionless mode*

2.2.3.2 The *connection mode*

2.2.3.2.1 Overview

The *connection mode* enables sending a byte stream reliable and in sequence. It provides an identification procedure, thus avoiding address resolution and transmission during the transmit stage.

This mode is often used for applications requiring a steady and reliable connection.

2.2.3.2.2 The four phases of the *connection mode*

The *connection mode* can be divided into four phases:

- The *local management phase*

During this phase, the local operations between user and provider are defined. The *T_OPEN* routine for example opens a channel of communication with the transport provider. In addition, a transport address is associated with a transport end point and each user may invoke several different channels between user and provider.

A list of the functions is to be found in the appendix, section 3.5.

- The *connection establishment* phase

This phase establishes a connection between two users. A Client-Server connection is built up. As usual, the Server provides the service to the network, and the user connects to the server in order to receive information (see section 2.2.2, *Types of messages between user and provider*).

A list of the functions is to be found in the appendix, section 3.6.

- The *data transfer* phase

The *data transfer phase* enables data transfer in both directions (bidirectional) within an established connection.
The data is sent in sequenced order.

A list of the functions is to be found in the appendix, section
3.7.

- The *connection release* phase

This phase enables to break an established connection. The user may ask the provider to release the transport connection (orderly or abortive release). The orderly release terminates the communication gracefully without loss of data, whereas the abortive release forces an immediate break.

A list of the functions is to be found in the appendix, section 3.8.

2.2.3.3 The *connectionless mode*

2.2.3.3.1 Overview

The *connectionless mode* is message oriented and sends data without any logical linkage between the individual data packets. In this mode, each packet of data contains all the necessary information about the data to be sent, for example the destination address.

This mode is often used for short term requests and responses, when a sequenced delivery of data is not obligatory.

2.2.3.3.2 The two phases of the *connectionless mode*

- The *local management phase*

This phase is equal to the *local management phase* of the *connection mode* , see section 2.2.3.2.2.

- The *data transfer phase*

The *data transfer phase* enables the user to transfer data units to the specified user of the same level. Each data unit has to contain the transport address of the destination user.

A list of the functions is to be found in the appendix, section 3.9.

3) Appendix

3.1 STREAMS System Calls

The STREAMS-related system calls are:

open(2) open a stream
close(2) close a stream
read(2) read data from a stream
write(2) write data to a stream
ioctl(2) control a stream
getmsg(2) receive a message at the stream head
putmsg(2) send a message downstream
poll(2) notify the application program when selected events occur on a Stream
pipe(2) create a channel that provides a communication path between multiple processes

3.2 Extracts on streams from the UNIX online manual

Following some extracts from the UNIX online manual providing special information for programmers:

- ioctl, which perform a variety of control functions on devices and streams; specific STREAMS functions provided by:
- STREAMS ioctl commands, a subset of the ioctl system call
- open(2) in connection with streams
- close(2) in connection with streams

NAME

ioctl - control device

SYNOPSIS

```
#include <unistd.h>

int ioctl (int fildes, int request, ... /* arg */);
```

DESCRIPTION

ioctl performs a variety of control functions on devices and STREAMS. For non-STREAMS files, the functions performed by this call are device-specific control functions. "request" and an optional third argument with varying type are passed to the file designated by fildes and are interpreted by the device driver. This control is rarely used on non-STREAMS devices, where the basic input/output functions are usually performed through the read(2) and write(2) system calls.

For STREAMS files, specific functions are performed by the ioctl call as described in streamio(7). For ufs files see ufsioctl(7/ICL).

...

SEE ALSO

streamio(7) in the Programmer's Guide: STREAMS.
termio(7) and ufsioctl(7/ICL) in the System Administrator's Reference Manual.

DIAGNOSTICS

Upon successful completion, the value returned depends upon the device control function, but must be a non-negative integer. Otherwise, a value of -1 is returned and errno is set to indicate the error.

...

NAME

streamio - STREAMS ioctl commands

SYNOPSIS

```
#include <sys/types.h>
#include <stropts.h>

int ioctl (int fildes, int command, ... /* arg */);
```

DESCRIPTION

STREAMS [see intro(2)] ioctl commands are a subset of the ioctl(2) system calls which perform a variety of control functions on streams.

- fildes is an open file descriptor that refers to a stream.
- command determines the control function to be performed as described below.
- arg represents additional information that is needed by this command.

The type of arg depends upon the command, but it is generally an integer or a pointer to a command-specific data structure. The command and arg are interpreted by the stream head. Certain combinations of these arguments may be passed to a module or driver in the stream.

Since these STREAMS commands are a subset of ioctl, they are subject to the errors described there. In addition to those errors, the call will fail with errno set to EINVAL, without processing a control function, if the stream referenced by fildes is linked below a multiplexor, or if command is not a valid value for a stream.

Also, as described in ioctl, STREAMS modules and drivers can detect errors. In this case, the module or driver sends an error message to the stream head containing an error value. This causes subsequent system calls to fail with errno set to this value.

COMMAND FUNCTIONS

The following ioctl commands, with error values indicated, are applicable to all STREAMS files:

I_PUSH Pushes the module whose name is pointed to by arg onto the top of the current stream, just below the stream head. If the stream is a pipe, the module will be inserted between the stream heads of both ends of the pipe. It then calls the open routine of the newly-pushed module. On failure, errno is set to one of the following values:

EINVAL Invalid module name.

EFAULT arg points outside the allocated

address space.

ENXIO Open routine of new module failed.

ENXIO Hangup received on fildes.

I_POP Removes the module just below the stream head of the stream pointed to by fildes. To remove a module from a pipe requires that the module was pushed on the side it is being removed from. arg should be 0 in an I_POP request. On failure, errno is set to one of the following values:

EINVAL No module present in the stream.

ENXIO Hangup received on fildes.

I_LOOK Retrieves the name of the module just below the stream head of the stream pointed to by fildes, and places it in a null terminated character string pointed at by arg. The buffer pointed to by arg should be at least FMNAMESZ+1 bytes long. An (#include <sys/conf.h>) declaration is required. On failure, errno is set to one of the following values:

EFAULT arg points outside the allocated address space.

EINVAL No module present in stream.

I_FLUSH This request flushes all input and/or output queues, depending on the value of arg. Legal arg values are:

FLUSHR Flush read queues.

FLUSHW Flush write queues.

FLUSHRW Flush read and write queues.

If a pipe or FIFO does not have any modules pushed, the read queue of the stream head on either end is flushed depending on the value of arg.

If FLUSHR is set and fildes is a pipe, the read queue for that end of the pipe is flushed and the write queue for the other end is flushed. If fildes is a FIFO, both queues are flushed.

If FLUSHW is set and fildes is a pipe and the other end of the pipe exists, the read queue for the other end of the pipe is flushed and the write queue for this end is flushed. If fildes is a FIFO, both queues of the FIFO are

flushed.

If FLUSHRW is set, all read queues are flushed, that is, the read queue for the FIFO and the read queue on both ends of the pipe are flushed.

Correct flush handling of a pipe or FIFO with modules pushed is achieved via the pipemod module. This module should be the first module pushed onto a pipe so that it is at the midpoint of the pipe itself.

On failure, errno is set to one of the following values:

ENOSR Unable to allocate buffers for flush message due to insufficient STREAMS memory resources.

EINVAL Invalid arg value.

ENXIO Hangup received on fildes.

I_FLUSHBAND Flushes a particular band of messages. arg points to a bandinfo structure that has the following members:

```
unsigned char   bi_pri;
int             bi_flag;
```

The bi_flag field may be one of FLUSHR, FLUSHW, or FLUSHRW as described earlier.

I_SETSIG Informs the stream head that the user wishes the kernel to issue the SIGPOLL signal [see signal(2)] when a particular event has occurred on the stream associated with fildes. I_SETSIG supports an asynchronous processing capability in STREAMS. The value of arg is a bitmask that specifies the events for which the user should be signaled. It is the bitwise-OR of any combination of the following constants:

S_INPUT Any message other than an M_PCPROTO has arrived on a stream head read queue. This event is maintained for compatibility with prior UNIX System V releases. This is set even if the message is of zero length.

S_RDNORM An ordinary (non-priority) message has arrived on a stream head read queue. This is set even if the message is of zero length.

S_RDBAND A priority band message (band > 0) has arrived on a stream head read

queue. This is set even if the message is of zero length.

S_HIPRI A high priority message is present on the stream head read queue. This is set even if the message is of zero length.

S_OUTPUT The write queue just below the stream head is no longer full. This notifies the user that there is room on the queue for sending (or writing) data downstream.

S_WRNORM This event is the same as S_OUTPUT.

S_WRBAND A priority band greater than 0 of a queue downstream exists and is writable. This notifies the user that there is room on the queue for sending (or writing) priority data downstream.

S_MSG A STREAMS signal message that contains the SIGPOLL signal has reached the front of the stream head read queue.

S_ERROR An M_ERROR message has reached the stream head.

S_HANGUP An M_HANGUP message has reached the stream head.

S_BANDURG When used in conjunction with S_RDBAND, SIGURG is generated instead of SIGPOLL when a priority message reaches the front of the stream head read queue.

A user process may choose to be signaled only of high priority messages by setting the arg bitmask to the value S_HIPRI.

Processes that wish to receive SIGPOLL signals must explicitly register to receive them using I_SETSIG. If several processes register to receive this signal for the same event on the same stream, each process will be signaled when the event occurs.

If the value of arg is zero, the calling process will be unregistered and will not receive further SIGPOLL signals. On failure, errno is set to one of the following values:

EINVAL — arg value is invalid or arg is zero and process is not registered to receive the SIGPOLL signal.

EAGAIN — Allocation of a data structure to store the signal request failed.

I_GETSIG — Returns the events for which the calling process is currently registered to be sent a SIGPOLL signal. The events are returned as a bitmask pointed to by arg, where the events are those specified in the description of I_SETSIG above. On failure, errno is set to one of the following values:

EINVAL — Process not registered to receive the SIGPOLL signal.

EFAULT — arg points outside the allocated address space.

I_FIND — Compares the names of all modules currently present in the stream to the name pointed to by arg, and returns 1 if the named module is present in the stream. It returns 0 if the named module is not present. On failure, errno is set to one of the following values:

EFAULT — arg points outside the allocated address space.

EINVAL — arg does not contain a valid module name.

I_PEEK — Allows a user to retrieve the information in the first message on the stream head read queue without taking the message off the queue. I_PEEK is analogous to getmsg(2) except that it does not remove the message from the queue. arg points to a strpeek structure which contains the following members:

```
struct strbuf ctlbuf;
struct strbuf databuf;
long flags;
```

The maxlen field in the ctlbuf and databuf strbuf structures [see getmsg(2)] must be set to the number of bytes of control information and/or data information, respectively, to retrieve. flags may be set to RS_HIPRI or 0. If RS_HIPRI is set, I_PEEK will look for a high priority message on the stream head read queue. Otherwise, I_PEEK will look for the first message on the stream head read queue.

I_PEEK returns 1 if a message was retrieved, and returns 0 if no message was found on the stream head read queue. It does not wait for a

message to arrive. On return, ctlbuf specifies information in the control buffer, databuf specifies information in the data buffer, and flags contains the value RS_HIPRI or 0. On failure, errno is set to the following value:

EFAULT	arg points, or the buffer area specified in ctlbuf or databuf is, outside the allocated address space.
EBADMSG	Queued message to be read is not valid for I_PEEK
EINVAL	Illegal value for flags.

I_SRDOPT Sets the read mode [see read(2)] using the value ofthe argument arg. Legal arg values are:

RNORM	Byte-stream mode, the default.
RMSGD	Message-discard mode.
RMSGN	Message-nondiscard mode.

In addition, treatment of control messages by the stream head may be changed by setting the following flags in arg:

RPROTNORM	Fail read() with EBADMSG if a control message is at the front of the stream head read queue. This is the default behavior.
RPROTDAT	Deliver the control portion of a message as data when a user issues read().
RPROTDIS	Discard the control portion of a message, delivering any data portion, when a user issues a read().

On failure, errno is set to the following value:

EINVAL	arg is not one of the above legal values.

I_GRDOPT Returns the current read mode setting in an int pointed to by the argument arg. Read modes are described in read(2). On failure, errno is set to the following value:

EFAULT	arg points outside the allocated address space.

I_NREAD Counts the number of data bytes in data blocks in the first message on the stream head read queue, and places this value in the location pointed to by arg. The return value for the command is the number of messages on the stream head read queue. For example, if zero is returned in arg, but the ioctl return value is greater than zero, this indicates that a zero-length message is next on the queue. On failure, errno is set to the following value:

EFAULT arg points outside the allocated address space.

I_FDINSERT Creates a message from user specified buffer(s), adds information about another stream and sends the message downstream. The message contains a control part and an optional data part. The data and control parts to be sent are distinguished by placement in separate buffers, as described below.

arg points to a strfdinsert structure which contains the following members:

```
struct strbufctlbuf;
struct strbufdatabuf;
long flags;
int fildes;
int offset;
```

The len field in the ctlbuf strbuf structure [see putmsg(2)] must be set to the size of a pointer plus the number of bytes of control information to be sent with the message. fildes in the strfdinsert structure specifies the file descriptor of the other stream. offset, which must be word-aligned, specifies the number of bytes beyond the beginning of the control buffer where I_FDINSERT will store a pointer. This pointer will be the address of the read queue structure of the driver for the stream corresponding to fildes in the strfdinsert structure. The len field in the databuf strbuf structure must be set to the number of bytes of data information to be sent with the message or zero if no data part is to be sent.

flags specifies the type of message to be created. An ordinary (non-priority) message is created if flags is set to 0, a high priority message is created if flags is set to RS_HIPRI. For normal messages, I_FDINSERT will block if the stream write queue is full due to internal flow control conditions. For high priority messages, I_FDINSERT does not block on this condition. For normal messages, I_FDINSERT

does not block when the write queue is full and O_NDELAY or O_NONBLOCK is set. Instead, it fails and sets errno to EAGAIN.

I_FDINSERT also blocks, unless prevented by lack of internal resources, waiting for the availability of message blocks, regardless of priority or whether O_NDELAY or O_NONBLOCK has been specified. No partial message is sent. On failure, errno is set to one of the following values:

EAGAIN A non-priority message was specified, the O_NDELAY or O_NONBLOCK flag is set, and the stream write queue is full due to internal flow control conditions.

ENOSR Buffers could not be allocated for the message that was to be created due to insufficient STREAMS memory resources.

EFAULT arg points, or the buffer area specified in ctlbuf or databuf is, outside the allocated address space.

EINVAL One of the following: fildes in the strfdinsert structure is not a valid, open stream file descriptor; the size of a pointer plus offset is greater than the len field for the buffer specified through ctlptr; offset does not specify a properly-aligned location in the data buffer; an undefined value is stored in flags.

ENXIO Hangup received on fildes of the ioctl call or fildes in the strfdinsert structure.

ERANGE The len field for the buffer specified through databuf does not fall within the range specified by the maximum and minimum packet sizes of the topmost stream module, or the len field for the buffer specified through databuf is larger than the maximum configured size of the data part of a message, or the len field for the buffer specified through ctlbuf is larger than the maximum configured size of the control

part of a message.

I_FDINSERT can also fail if an error message was received by the stream head of the stream corresponding to fildes in the strfdinsert structure. In this case, errno will be set to the value in the message.

I_STR Constructs an internal STREAMS ioctl message from the data pointed to by arg, and sends that message downstream.

This mechanism is provided to send user ioctl requests to downstream modules and drivers. It allows information to be sent with the ioctl, and will return to the user any information sent upstream by the downstream recipient. I_STR blocks until the system responds with either a positive or negative acknowledgement message, or until the request "times out" after some period of time. If the request times out, it fails with errno set to ETIME.

At most, one I_STR can be activeon a stream. Further I_STR calls will block until the active I_STR completes at the stream head. The default timeout interval for these requests is 15 seconds. The O_NDELAY and O_NONBLOCK [see open(2)] flags have no effect on this call.

To send requests downstream, arg must point to a strioctl structure which contains the following members:

```
int     ic_cmd;
int     ic_timout;
int     ic_len;
char *ic_dp;
```

ic_cmd is the internal ioctl command intended for a downstream module or driver and ic_timout is the number of seconds (-1 = infinite, 0 = use default, >0 = as specified) an I_STR request will wait for acknowledgement before timing out. The default timeout is infinite. ic_len is the number of bytes in the data argument and ic_dp is a pointer to the data argument. The ic_len field has two uses: on input, it contains the length of the data argument passed in, and on return from the command, it contains the number of bytes being returned to the user (the buffer pointed to by ic_dp should be large enough to contain the maximum amount of data that any module or the driver in the stream can return).

The stream head will convert theinformation pointed to by the strioctl structure to an

internal ioctl command message and send it downstream. On failure, errno is set to one of the following values:

ENOSR — Unable to allocate buffers for the ioctl message due to insufficient STREAMS memory resources.

EFAULT — arg points, or the buffer area specified by ic_dp and ic_len (separately for data sent and data returned) is, outside the allocated address space.

EINVAL — ic_len is less than 0 or ic_len is larger than the maximum configured size of the data part of a message or ic_timout is less than -1.

ENXIO — Hangup received on fildes.

ETIME — A downstream ioctl timed out before acknowledgement was received.

An I_STR can also fail while waiting for an acknowledgement if a message indicating an error or a hangup is received at the stream head. In addition, an error code can be returned in the positive or negative acknowledgement message, in the event the ioctl command sent downstream fails. For these cases, I_STR will fail with errno set to the value in the message.

I_SWROPT — Sets the write mode using the value of the argument arg. Legal bit settings for arg are:

SNDZERO — Send a zero-length message downstream when a write of 0 bytes occurs.

SNDPIPE — Send a sigpipe signal to the process if it tries to write to the stream and there has been a previous write error on it.

To not send a zero-length message when a write of 0 bytes occurs or to not send a sigpipe signal after a previous write error, the relevant bits must not be set in arg.

On failure, errno may be set to the following value: On failure, errno may be set to the following value:

EINVAL — arg is the above legal value.

I_GWROPT Returns the current write mode setting, as described above, in the int that is pointed to by the argument arg.

I_SENDFD Requests the stream associated with fildes to send a message, containing a file pointer, to the stream head at the other end of a stream pipe. The file pointer corresponds to arg, which must be an open file descriptor.

I_SENDFD converts arg into the corresponding system file pointer. It allocates a message block and inserts the file pointer in the block. The user id and group id associated with the sending process are also inserted. This message is placed directly on the read queue [see intro(2)] of the stream head at the other end of the stream pipe to which it is connected. On failure, errno is set to one of the following values:

EAGAIN The sending stream is unable to allocate a message block to contain the file pointer.

EAGAIN The read queue of the receiving stream head is full and cannot accept the message sent by I_SENDFD.

EBADF arg is not a valid, open file descriptor.

EINVAL fildes is not connected to a stream pipe.

ENXIO Hangup received on fildes.

I_RECVFD Retrieves the file descriptor associated with the message sent by an I_SENDFD ioctl over a stream pipe. arg is a pointer to a data buffer large enough to hold an strrecvfd data structure containing the following members:

```
int    fd;
uid_t uid;
gid_t gid;
char fill[8];
```

fd is an integer file descriptor. uid and gid are the user id and group id, respectively, of the sending stream.

If O_NDELAY and O_NONBLOCK are clear [see open(2)], I_RECVFD will block until a message is present at the stream head. If O_NDELAY or O_NONBLOCK is set, I_RECVFD will fail with

errno set to EAGAIN if no message is present at the stream head.

If the message at the stream head is a message sent by an I_SENDFD, a new user file descriptor is allocated for the file pointer contained in the message. The new file descriptor is placed in the fd field of the strrecvfd structure. The structure is copied into the user data buffer pointed to by arg. On failure, errno is set to one of the following values:

EAGAIN A message is not present at the stream head read queue, and the O_NDELAY or O_NONBLOCK flag is set.

EBADMSG The message at the stream head read queue is not a message containing a passed file descriptor.

EFAULT arg points outside the allocated address space.

EMFILE NOFILES file descriptors are currently open.

ENXIO Hangup received on fildes.

EOVERFLOW uid or gid is too large to be stored in the structure pointed to by arg.

I_LIST Allows the user to list all the module names on the stream, up to and including the topmost driver name. If arg is NULL, the return value is the number of modules, including the driver, that are on the stream pointed to by fildes. This allows the user to allocate enough space for the module names. If arg is non-NULL, it should point to an str_list structure that has the following members:

```
int          sl_nmods;
struct str_mlist  *sl_modlist;
```

The str_mlist structure has the following member:

```
char l_name[FMNAMESZ+1];
```

sl_nmods indicates the number of entries the user has allocated in the array and on return, sl_modlist contains the list of module names. The return value indicates the number of entries that have been filled in. On failure, errno may be set to one of the following values:

EINVAL The sl_nmods member is less than 1.

EAGAIN Unable to allocate buffers

I_ATMARK Allows the user to see if the current message on the stream head read queue is "marked" by some module downstream. arg determines how the checking is done when there may be multiple marked messages on the stream head read queue. It may take the following values:

ANYMARK Check if the message is marked.

LASTMARK Check if the message is the last one marked on the queue.

The return value is 1 if the mark condition is satisfied and 0 otherwise. On failure, errno may be set to the following value:

EINVAL Invalid arg value.

I_CKBAND Check if the message of a given priority band exists on the stream head read queue. This returns 1 if a message of a given priority exists, or -1 on error. arg should be an integer containing the value of the priority band in question. On failure, errno may be set to the following value:

EINVAL Invalid arg value.

I_GETBAND Returns the priority band of the first message on the stream head read queue in the integer referenced by arg. On failure, errno may be set to the following value:

ENODATA No message on the stream head read queue.

I_CANPUT Check if a certain band is writable. arg is set to the priority band in question. The return value is 0 if the priority band arg is flow controlled, 1 if the band is writable, or -1 on error. On failure, errno may be set to the following value:

EINVAL Invalid arg value.

I_SETCLTIME Allows the user to set the time the stream head will delay when a stream is closing and there are data on the write queues. Before closing each module and driver, the stream head will delay for the specified amount of time to allow the data to drain. If, after the delay, data are still present, data will be flushed. arg

is a pointer to the number of milliseconds to delay, rounded up to the nearest legal value on the system. The default is fifteen seconds. On failure, errno may be set to the following value:

EINVAL Invalid arg value.

I_GETCLTIME Returns the close time delay in the long pointed by arg.

The following four commands are used for connecting and disconnecting multiplexed STREAMS configurations.

I_LINK Connects two streams, where fildes is the file descriptor of the stream connected to the multiplexing driver, and arg is the file descriptor of the stream connected to another driver. The stream designated by arg gets connected below the multiplexing driver. I_LINK requires the multiplexing driver to send an acknowledgement message to the stream head regarding the linking operation. This call returns a multiplexor ID number (an identifier used to disconnect the multiplexor, see I_UNLINK) on success, and a -1 on failure. On failure, errno is set to one of the following values:

ENXIO Hangup received on fildes.

ETIME Time out before acknowledgement message was received at stream head.

EAGAIN Temporarily unable to allocate storage to perform the I_LINK.

ENOSR Unable to allocate storage to perform the I_LINK due to insufficient STREAMS memory resources.

EBADF arg is not a valid, open file descriptor.

EINVAL fildes stream does not support multiplexing.

EINVAL arg is not a stream, or is already linked under a multiplexor.

EINVAL The specified link operation would cause a "cycle" in the resulting configuration; that is, if a given driver is linked into a multiplexing configuration in more

than one place.

EINVAL fildes is the file descriptor of a pipe or FIFO.

An I_LINK can also fail while waiting for the multiplexing driver to acknowledge the link request, if a message indicating an error or a hangup is received at the stream head of fildes. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, I_LINK will fail with errno set to the value in the message.

I_UNLINK Disconnects the two streams specified by fildes and arg. fildes is the file descriptor of the stream connected to the multiplexing driver. arg is the multiplexor ID number that was returned by the I_LINK. If arg is -1, then all Streams which were linked to fildes are disconnected. As in I_LINK, this command requires the multiplexing driver to acknowledge the unlink. On failure, errno is set to one of the following values:

ENXIO Hangup received on fildes.

ETIME Time out before acknowledgement message was received at stream head.

ENOSR Unable to allocate storage to perform the I_UNLINK due to insufficient STREAMS memory resources.

EINVAL arg is an invalid multiplexor ID number or fildes is not the stream on which the I_LINK that returned arg was performed.

EINVAL fildes is the file descriptor of a pipe or FIFO.

An I_UNLINK can also fail while waiting for the multiplexing driver to acknowledge the link request, if a message indicating an error or a hangup is received at the stream head of fildes. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, I_UNLINK will fail with errno set to the value in the message.

I_PLINK Connects two streams, where fildes is the file descriptor of the stream connected to the

multiplexing driver, and arg is the file descriptor of the stream connected to another driver. The stream designated by arg gets connected via a persistent link below the multiplexing driver. I_PLINK requires the multiplexing driver to send an acknowledgement message to the stream head regarding the linking operation. This call creates a persistent link which can exist even if the file descriptor fildes associated with the upper stream to the multiplexing driver is closed. This call returns a multiplexor ID number (an identifier that may be used to disconnect the multiplexor, see I_PUNLINK) on success, and a -1 on failure. On failure, errno may be set to one of the following values:

ENXIO Hangup received on fildes.

ETIME Time out before acknowledgement message was received at the stream head.

EAGAIN Unable to allocate STREAMS storage to perform the I_PLINK.

EBADF arg is not a valid, open file descriptor.

EINVAL fildes does not support multiplexing.

EINVAL arg is not a stream or is already linked under a multiplexor.

EINVAL The specified link operation would cause a "cycle" in the resulting configuration; that is, if a given stream head is linked into a multiplexing configuration in more than one place.

EINVAL fildes is the file descriptor of a pipe or FIFO.

An I_PLINK can also fail while waiting for the multiplexing driver to acknowledge the link request, if a message indicating an error on a hangup is received at the stream head of fildes. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, I_PLINK will fail with errno set to the value in the message.

I_PUNLINK Disconnects the two streams specified by fildes

and arg that are connected with a persistent link. fildes is the file descriptor of the stream connected to the multiplexing driver. arg is the multiplexor ID number that was returned by I_PLINK when a stream was linked below the multiplexing driver. If arg is MUXID_ALL then all streams which are persistent links to fildes are disconnected. As in I_PLINK, this command requires the multiplexing driver to acknowledge the unlink. On failure, errno may be set to one of the following values:

ENXIO Hangup received on fildes.

ETIME Time out before acknowledgement message was received at the stream head.

EAGAIN Unable to allocate buffers for the acknowledgement message.

EINVAL Invalid multiplexor ID number.

EINVAL fildes is the file descriptor of a pipe or FIFO.

An I_PUNLINK can also fail while waiting for the multiplexing driver to acknowledge the link request if a message indicating an error or a hangup is received at the stream head of fildes. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, I_PUNLINK will fail with errno set to the value in the message.

SEE ALSO

close(2), fcntl(2), getmsg(2), intro(2), ioctl(2), open(2), poll(2), putmsg(2), read(2), signal(2), write(2), signal(5). Programmer's Reference Manual.

DIAGNOSTICS

Unless specified otherwise above, the return value from ioctl is 0 upon success and -1 upon failure with errno set as indicated.

NAME

open - open file for reading or writing

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open (const char *path, int oflag, ... /*          mode_t mode
*/);
```

DESCRIPTION

path points to a pathname naming a file. open opens a file descriptor for the named file and sets the file status flags according to the value of oflag. oflag values are constructed by OR-ing Flags from the following list (only one of the first three flags below may be used):

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing.

...

When opening a STREAMS file, oflag may be constructed from O_NDELAY or O_NONBLOCK OR-ed with either O_RDONLY, O_WRONLY , or O_RDWR. Other flag values are not applicable to STREAMS devices and have no effect on them. The values of O_NDELAY and O_NONBLOCK affect the operation of STREAMS drivers and certain system calls [see read(2), getmsg(2), putmsg(2), and write(2)]. For drivers, the implementation of O_NDELAY and O_NONBLOCK is device specific. Each STREAMS device driver may treat these options differently.

When open is invoked to open a named stream, and the connld module [see connld(7)] has been pushed on the pipe, open blocks until the server process has issued an I_RECVFD ioctl [see streamio(7)] to receive the file descriptor.

If path is a symbolic link and O_CREAT and O_EXCL are set, the link is not followed.

The file pointer used to mark the current position within the file is set to the beginning of the file.

The new file descriptor is the lowestnumbered file descriptor available and is set to remain open across exec system calls [see fcntl(2)].

Certain flag values can be set following open as described in fcntl(2).

If O_CREAT is set and the file did not previously exist, upon successful completion open marks for update the st_atime, st_ctime and st_mtime fields of the file and the st_ctime and st_mtime fields of the parent directory.

If O_TRUNC is set and the file did previously exist, upon successful completion open marks for update the st_ctime and st_mtime fields of the file.

The named file is opened unless one or more of the following are true:

EACCES The file does not exist and write permission is denied by the parent directory of the file to be created.

EACCES O_TRUNC is specified and write permission is denied.

EACCES A component of the path prefix denies search permission.

EACCES oflag permission is denied for an existing file.

EAGAIN The file exists, mandatory file/record locking is set, and there are outstanding record locks on the file [see chmod(2)].

EEXIST O_CREAT and O_EXCL are set, and the named file exists.

EFAULT path points outside the allocated address space of the process.

EINTR A signal was caught during the open system call.

EIO A hangup or error occurred during the open of the STREAMS-based device.

EISDIR The named file is a directory and oflag is write or read/write.

ELOOP Too many symbolic links were encountered in translating path.

EMFILE The process has too many open files [see getrlimit(2)].

EMULTIHOP Components of path require hopping to multiple remote machines and the file system does not allow it.

ENAMETOOLONG The length of the path argument exceeds {PATH_MAX}, or the length of a path component exceeds {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.

ENFILE The system file table is full.

ENOENT O_CREAT is not set and the named file does not exist.

ENOENT O_CREAT is set and a component of the path prefix does not exist or is the null pathname.

ENOLINK path points to a remote machine, and the link to that machine is no longer active.

ENOMEM The system is unable to allocate a send descriptor.

ENOSPC O_CREAT and O_EXCL are set, and the file system is out of inodes.

ENOSPC O_CREAT is set and the directory that would contain the file cannot be extended.

ENOSR Unable to allocate a stream.

ENOTDIR A component of the path prefix is not a directory.

ENXIO The named file is a character special or block special file, and the device associated with this special file does not exist.

ENXIO O_NDELAY or O_NONBLOCK is set, the named file is a FIFO, O_WRONLY is set, and no process has the file open for reading.

ENXIO A STREAMS module or driver open routine failed.

EROFS The named file resides on a read-only file system and either O_WRONLY, O_RDWR, O_CREAT, or O_TRUNC is set in oflag (if the file does not exist).

ETXTBSY The file is a pure procedure (shared text) file that is being executed and oflag is write or read/write.

SEE ALSO

intro(2), chmod(2), close(2), creat(2), dup(2), exec(2),

fcntl(2), getrlimit(2), lseek(2), read(2), getmsg(2), putmsg(2), stat(2), umask(2), write(2), stat(5).

DIAGNOSTICS

Upon successful completion, the file descriptor is returned. Otherwise, a value of -1 is returned and errno is set to indicate the error.

NAME

close - close a file descriptor

SYNOPSIS

```
#include <unistd.h>
int close(int fildes);
```

DESCRIPTION

fildes is a file descriptor obtained from a creat, open, dup, fcntl, or pipe system call. close closes the file descriptor indicated by fildes. All outstanding record locks owned by the process (on the file indicated by fildes) are removed.

When all file descriptors associated with the open file description have been closed, the open file description is freed.

If the link count of the file is zero, when all file descriptors associated with the file have been closed, the space occupied by the file is freed and the file is no longer accessible.

If a STREAMS-based [see intro(2)] fildes is closed, and the calling process had previously registered to receive a SIGPOLL signal [see signal(2)] for events associated with that stream [see I_SETSIG in streamio(7)], the calling process will be unregistered for events associated with the stream. The last close for a stream causes the stream associated with fildes to be dismantled. If O_NDELAY and O_NONBLOCK are clear and there have been no signals posted for the stream, and if there are data on the module's write queue, close waits up to 15 seconds (for each module and driver) for any output to drain before dismantling the stream. The time delay can be changed via an I_SETCLTIME ioctl request [see streamio(7)]. If O_NDELAY or O_NONBLOCK is set, or if there are any pending signals, close does not wait for output to drain, and dismantles the stream immediately.

If fildes is associated with one end of a pipe, the last close causes a hangup to occur on the other end of the pipe. In addition, if the other end of the pipe has been named [see fattach(3C)], the last close forces the named end to be detached [see fdetach(3C)]. If the named end has no open processes associated with it and becomes detached, the stream associated with that end is also dismantled.

The named file is closed unless one or more of the following are true:

EBADF fildes is not a valid open file descriptor.

EINTR A signal was caught during the close system call.

ENOLINK fildes is on a remote machine and the link to that machine is no longer active.

SEE ALSO

creat(2), dup(2), exec(2), fcntl(2), intro(2), open(2), pipe(2), signal(2), signal(5).
streamio(7) in the Programmer's Guide: STREAMS.
fattach(3C), fdetach(3C) in the Programmer's Guide: Networking Interfaces.

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and errno is set to indicate the error.

3.3 Socket system calls

SOCKET creates an end point for communications, returning a file descriptor

BIND binds the file descriptor to the socket, giving the socket its name

ACCEPT receives incoming requests for connection to the server and returns a file descriptor

CONNECT requests the kernel to make a connection to an existing socket

LISTEN specifies the maximum length of a queue

GETSOCKETNAME gets the name of a socket, bound by a previous BIND() call

GETSOCKETPT \ / retrieve and set options associated with

and > < sockets, according to communications

SETSOCKETPT / \ domain and protocols of the socket

SHUTDOWN closes a socket connection

CLOSE closes a socket

3.4 Possible values for *type* in the system call

socket(*domain*, *type*, *protocol*)

SOCK_STREAM provides reliable, sequenced, byte by byte communication similar to pipes. Transmission may by bidirectional.

SOCK_DGRAM is used in datagram transmission where the packets are of different length. Transmission again bidirectional.

SOCK_SEQPACKET datagrams of maximum length are bidirectionally and sequenced transmitted. With each read call the entire packet has to be read in full length.

SOCK_RAW for reliable transmission of the packets. Enables specifying the path to be used in the transmission.

3.5 List of functions within the *local management phase* of the *TLI connection mode*

T_BIND assigns addresses to each transport end point, similar to *bind* with sockets

T_ALLOC allocates transport level data structures

T_GETINFO returns a set of parameters associated with a particular transport provider

T_GETSTATE returns the state of a transport endpoint

T_LOOK returns the current event on a transport endpoint

T_OPTMGMT negotiates protocol-specific options with the transport provider

T_CLOSE closes a transport endpoint

T_ERROR prints a transport interface error message

T_FREE frees structures already allocated

T_OPEN establishes a transport endpoint connected to a chosen transport provider

T_SYNC synchronizes a transport endpoint with the transport provider

T_UNBIND unbinds a transport address from a transport end point

3.6 List of functions within the *connection establishment phase* of the *TLI connection mode*

T_ACCEPT accepts a request for a transport connection

T_CONNECT establishes a connection with the transport user at a specified destination

T_LISTEN retrives an indication of a current request from another transport user

T_REVCONNECT completes connection establishment if T_CONNECT had been called in asynchronous mode

3.7 List of functions within the *data transfer phase* of the *TLI connection mode*

T_RECV retrieves data that has arrived over the transport layer

T_SEND sends data across an established connection

3.8 List of functions within the *connection release phase* of the *TLI connection mode*

T_RCVDIS returns an indication of an aborted connection, including a reason code and user data

T_RCVREL returns an indication that the remote user has requested an orderly release of a connection

T_SNDDIS aborts a connection or rejects a connect request

T_SNDREL requests the orderly release of a connection

3.9 List of functions within the *data transfer phase* of the *TLI connectionless mode*

T_RCVUDATA retrieves a message sent by another transport user

T_RCVUDERR retrieves error information associated with a previously sent message

T_SNDUDATA sends a message to the specified destination user

3.10 Reference of used books and notes

- UNIX System V Release 4
  'Grundlagen und Praxis' - tewi, K. Rosen, R. Rosinski,
  J. Farber
- UNIX - online manual about streams, sockets etc.

In addition, I used some books and information from the technicians at RTC:

- Programmer's Guide : Streams
  DRS SYSTEMS NX V4.0 - ICL
  - The I/O Subsystem : Streams
  - Interprocess Communication : Sockets
  - Network Communications
  - Systems Programming : Communications