STREAMS Programmer's Guide

HP 9000 and Integrity Server Computer Systems

Edition 1

Manufacturing Part Number : 5991-4437
October 2005

United States

Legal Notices

Printing History

New editions of this manual will incorporate all material updated since the previous edition. The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (minor corrections that are incorporated at a reprinting do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

Edition/Part Number/Date

First/5991-4437/October 2005

Contents

Contents

Figures

About This Document

This document describes how to use STREAMS, a generalized, flexible communication framework and set of tools that facilitate the development of communication services for UNIX.

The document printing date and part number indicate the document's current edition. The printing date will change when a new edition is printed. Minor changes may be made at reprint without changing the printing date. The document part number will change when extensive changes are made.

Document updates may be issued between editions to correct errors or document product changes. To ensure that you receive the updated or new editions, you should subscribe to the appropriate product support service.

See your HP sales representative for details.

The manual is organized as follows:

Chapter 1, "Overview."

This chapter shows an overview of the STREAMS components, system calls, messages and queues and STREAMS multiplexor.

Chapter 2, "STREAMS Mechanism and System Calls."

This chapter describes system calls most commonly seen in user applications that interact with STREAMS devices.

Chapter 3, "Messages."

This chapter discusses STREAMS messages, their structure, linkage, queuing and interfacing into other STREAMS components.

Chapter 4, "Modules and Drivers."

Describes various data structures essential to modules and drivers and provides design guidelines for developing them.

Chapter 5, "Multiplexing."

Shows how the STREAMS IOCTL commands enable the user process to perform a variety of control functions on a stream.

Appendix A, "STREAMS IOCTL Commands."

Shows how the STREAMS IOCTL commands enable the user process to perform a variety of control functions on a stream.

Appendix B, "STREAMS Utilities Supported by HP-UX."

Deals with the STREAMS utilities supported by HP-UX which are used to perform specific operations/functions in module and driver development.

Appendix C, "Message Types."

Describes the fixed set of message types recognized by STREAMS/UX.

Appendix D, "STREAMS Administrative Driver."

Describes the STREAMS Administrative Driver. The STREAMS Administrative Driver, also known as SAD, is an interface to the autopush facility. SAD enables administrative tasks to be performed on STREAMS modules and drivers.

Appendix E, "Differences Between STREAMS / UX and System V Release 4 STREAMS."

Shows the differences between STREAMS/UX and System V Release 4.2 STREAMS.

Appendix F, "Synchronization Levels."

Explains the 5 levels of synchronization provided by STREAMS/UX.

Appendix G, "STREAMS Commands."

Describes STREAMS commands.

Appendix H, "STREAMS Kernel Tunable Parameters."

Describes the STREAMS kernel tunable parameters.

Intended Audience

This document is intended for HP-UX developers.

This document is not a tutorial.

The "Support/Compatibility Disclaimers" section describes the support provided by Hewlett-Packard Company.

Using This Manual

How you read this manual depends on the tasks you need to perform. The steps you need to take will differ depending on whether you are writing a new driver or porting an existing driver.

Because of page width restrictions, certain lines of STREAMS code exceed the space available and break in unintended places. Please treat these "broken" lines as one line.

HP-UX 11i v1 and HP-UX 11i v2 users will need to install STREAMS Advance Release (STAR) 1.0 to obtain NOSYNC functionality. Refer to the STREAMS Advance Release (STAR) 1.0 Release Notes at http://www.docs.hp.com/ for more information on STAR 1.0.

Typographical Conventions

This document uses the following conventions.

audit (5)	An HP-UX manpage. In this example, *audit* is the name and *5* is the section in the *HP-UX Reference*. On the web and on the Instant Information CD, it may be a hot link to the manpage itself. From the HP-UX command line, you can enter "`man audit`" or "`man 5 audit`" to view the manpage. See *man* (1).
Book Title	The title of a book. On the web and on the Instant Information CD, it may be a hot link to the book itself.
KeyCap	The name of a keyboard key. Note that **Return** and **Enter** both refer to the same key.
Emphasis	Text that is emphasized.
Bold	Text that is strongly emphasized.
Bold	The defined use of an important word or phrase.
`ComputerOut`	Text displayed by the computer.
`UserInput`	Commands and other text that you type.
`Command`	A command name or qualified command phrase.
`Variable`	The name of a variable that you may replace in a command or function or information in a display that represents several possible values.
`[ ]`	The contents are optional in formats and command descriptions. If the contents are a list separated by |, you must choose one of the items.
`{ }`	The contents are required in formats and command descriptions. If the contents are a list separated by |, you must choose one of the items.
...	The preceding element may be repeated an arbitrary number of times.
|	Separates items in a list of choices.

HP Encourages Your Comments

HP encourages your comments concerning this document. We are committed to providing documentation that meets your needs.

Send comments to: `netinfo_feedback@cup.hp.com`

Include the document title, manufacturing part number, and any comment, error found, or suggestion for improvement you have concerning this document.

Email and Internet Resources

Interface program and developer resource materials are available at the following locations:

- Develop Drivers for HP-UX at `http:/www.hp.com/go/hpux_ddk`
- Interface Program E-mail at `interface@fc.hp.com`
- Developer Resource at `http://www.hp.com/dspp`
- Hardware Provider Program at `http://www.hp.com/dspp/hphp`

Support and Compatibility Disclaimers

Since drivers function at the kernel level, Hewlett-Packard Company (HP) reminds you of the following:

Adding your own driver to HP-UX requires relinking the driver into HP-UX. With each new release, plan on recompiling the driver and reinstalling it in the new HP-UX kernel. Many header files do not change. However, drivers typically use some header files that could change across releases (that is, you can have some system dependencies).

HP provides support services for HP products, including HP-UX. Products, including drivers, from non-HP parties receive no support, other than the support of those parts of a driver that rely on the documented behavior of supported HP products.

If difficulties arise during the development and test phases of writing a driver, HP may provide assistance in isolating problems to determine if:

- HP hardware is not at fault; and
- HP software and firmware is not at fault by removing user-written kernel drivers.

Reference Documentation

- *UNIX SVR4.2 Command Reference Manual*
- *SVR4.2 STREAMS Manual*
- *UNIX SVR4.2 Operation System API Reference Manual*
- *UNIX System V Release 4 Programmer's Guide: STREAMS*
- *SVR4.2 Driver Manual*
- *UNIX SVR4.2 System Files and Devices Reference Manual*

1 Overview

STREAMS is a generalized flexible communication framework. STREAMS provides a set of tools to facilitate the development of communication services for UNIX. It simplifies the user application interface to character device drivers, and makes the application independent of the underlying implementation. The modularity and dynamic module selection features of STREAMS make it an effective framework for implementing and interacting with kernel level components. These components are device drivers, network controllers, and network protocols.

STREAMS was developed by Bell Labs in the early 1980s as an improvement on UNIX character I/O. Raw character I/O does not provide advanced features like buffer management or flow control. STREAMS supports all the basic functionality of character I/O, and provides additional features like dynamic module selection, queues, polling, multiplexing, and flow control. This makes STREAMS an optimal framework for writing networking protocols and related software, where performance, delivery and high availability are critical.

NOTE The term **stream** is used in multiple contexts when describing UNIX I/O. The term "Standard I/O Streams" refers to standard UNIX library calls consisting of *fopen* (), *fclose* (), *fread* (), and *fprint* (). **STREAMS** refers to the framework where a user application interacts with the UNIX kernel for communication services.

A stream is a full-duplex path between a user process and a device driver or pseudo-driver, in the kernel space. A pseudo-driver is a software component that emulates driver functionality, but does not manage a real hardware device. STREAMS applications use of a set of system calls to open a driver, write messages to the driver, and read the processed messages back from the driver. The stream implements a connection between the kernel and one or more user processes, and shares the properties of STREAMS-based devices.

STREAMS Components

A stream consists of the following components shown in *Figure 1-1, "Stream with a STREAMS Module."*Figure 1-2 on page 25 shows a stream without a STREAMS module.

- Stream head
- Modules (optional)
- Target device driver or pseudo-driver

Figure 1-1 Stream with a STREAMS Module

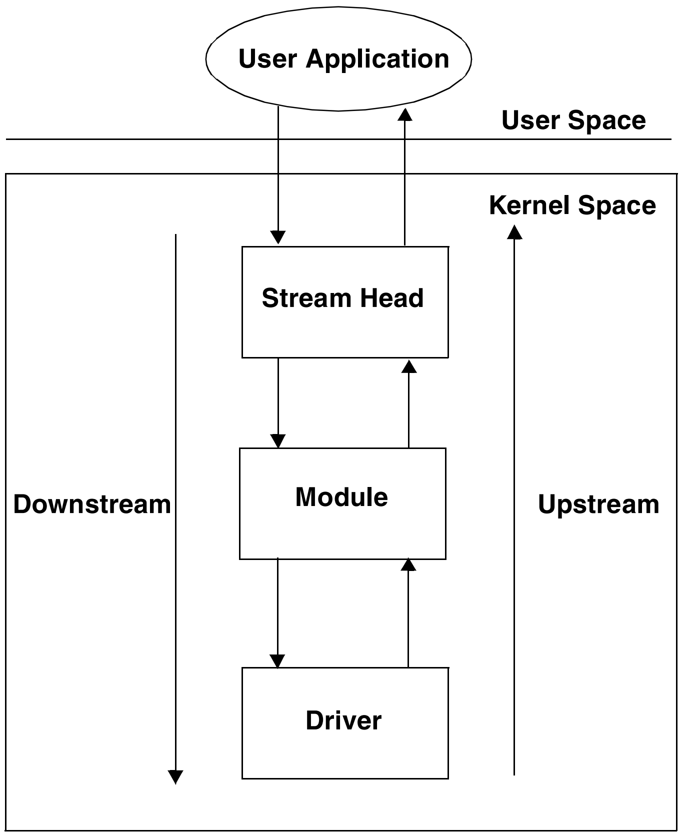
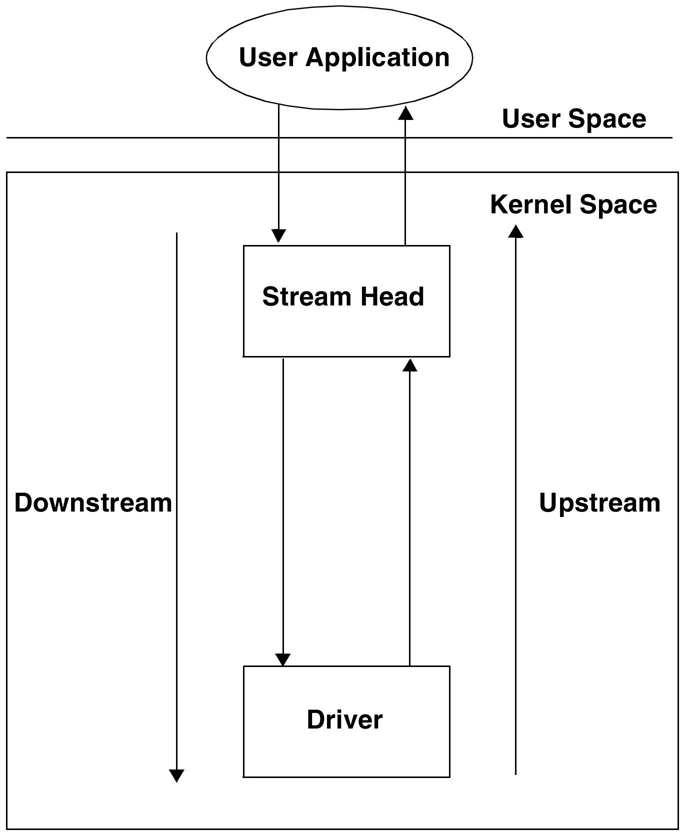


Figure 1-2 Stream without a STREAMS Module



Stream Head

The stream head is the interface between the user process and the STREAMS driver or module. It consists of a set of data structures and associated kernel routines. The kernel routines operate on the data structures and interface with the next stream component in the sequence. The stream head receives data from user processes, packs it into STREAMS messages, and sends these messages downstream to the module or driver. The stream head also receives data back from the module or driver, and makes the data available for the user process.

STREAMS Module

Modules are optional components of a stream. Modules process the messages in a stream before sending them to the driver. The functionality provided by a module is based on application requirements. Message encryption, decryption, and string manipulation are some of the examples of module functionality. A user process pushes a module on to a stream via the *ioctl*() system call. The new module becomes the first component below the stream head. When multiple modules are pushed on to a stream, the last module to be pushed is located just below the stream head. Figure 1-1, "Stream with a STREAMS Module," shows a block diagram of a stream with one module. Figure 1-2, "Stream without a STREAMS Module," shows a block diagram of a stream without modules.

STREAMS Driver or Pseudo-Driver

A STREAMS driver is located below the stream head and modules and is the stream end. The driver can act on a real external I/O device. Or, it can be a pseudo-driver.

Messages and Queues

This section introduces the queues and messages associated with STREAMS.

Queues

Each stream component is associated with a read queue and write queue pair. The read queue and write queue are the data structures used to record the status of the stream component. These queues record the messages to be processed later.

STREAMS allocates the read queue and write queue to the stream component. The allocation is performed when the component becomes part of a stream. For example, a module's read queue and write queue are allocated when the module is pushed on the stream.

When a stream component is removed from the stream, STREAMS frees its corresponding read queue and write queue. For example, a module's read queue and write queue are freed when the module is popped out of a stream.

Messages

Communication in STREAMS is based on messages. STREAMS modules and drivers communicate with each other by passing pointers to these messages. Every message consists of the following data structures:

msgb	Describes the message type.
datab	Contains a pointer to the message data and other message details.

Messages in a stream are passed from one module or driver to another by invoking the *put* procedure of the next module or driver. For details on the *put* procedure, see *Chapter 3, "Messages,"* and *Chapter 4, "Modules and Drivers,"*

Messages that are not being processed are queued in a linked list of messages called the message queue. The head and tail of the message queue are included in the queue itself.

Message Types

Each message is assigned a message type upon creation. The two message types are Normal (ordinary) and High Priority. The message type is used by modules and drivers to determine the type of message processing required. Although a module or driver usually assigns the message type to a message that it generates, a module can also change the message type while processing the message.

Message Priority

The priority of a message determines the order in which it is placed or processed in a queue.

To control the processing priority of normal messages on a queue, STREAMS supports the concept of the **priority band**. Priority bands are used to determine the order in which normal messages in a queue will be stored and processed. A priority band value ranging from 1 to 255 can be assigned to these messages.

Message Processing

Each queue is associated with a `put` procedure and an optional `service` procedure to process the messages.

The *put* procedure is used to process the messages from the preceding queue in a stream. Depending on the message type and the availability of the next module, the `put` procedure can consume this message and pass it to the next queue for further processing. Alternatively, the `put` procedure may place the messages on its message queue for processing later.

The `service` procedure is used to process the messages in the message queue. A queue will always contain a `put` procedure, but not necessarily a `service` procedure. Depending on the nature of the module, the module designer determines whether a `service` procedure is necessary. If the `put` procedure allows the messages to be queued on its own message queue, a corresponding `service` procedure should be provided to process these messages.

See Chapter 4 for more information on the `put` and `service` procedures.

A limit can be set for each queue. This limit is the maximum amount of data that can be queued in the message queue. This data flow control mechanism to allow the STREAMS modules to control the amount of data flow in the stream. HP recommends using data flow control for modules or drivers.

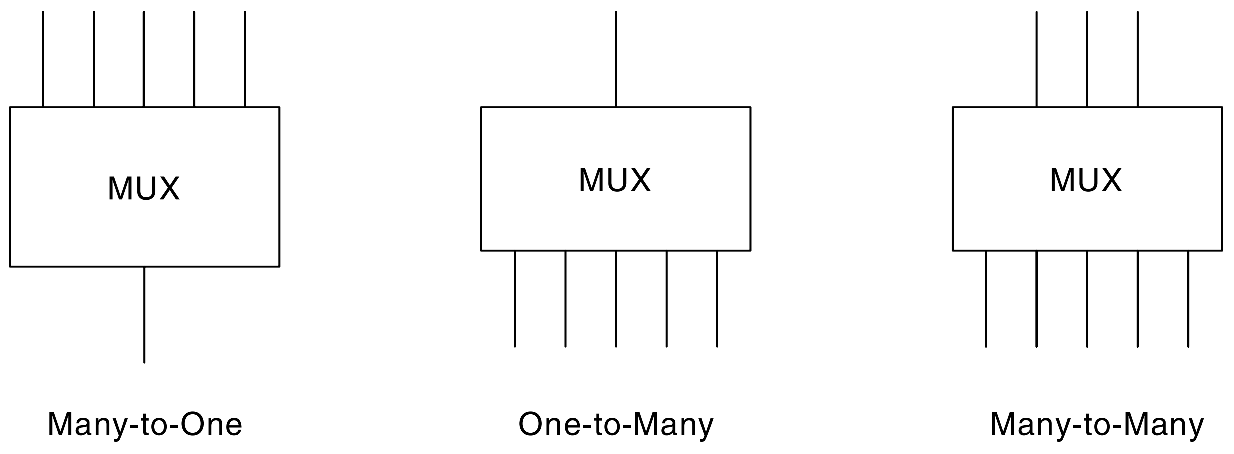
STREAMS Multiplexor

A stream is a set of components with messages flowing between them. Each module is connected to no more than one downstream module and one upstream module. Advanced applications can require more complex configurations than a one-to-one relationship. For instance, multiple modules can communicate with a driver. Or, a single module may need to route messages to multiple drivers.

The STREAMS multiplexor supports complex configurations between user processes, modules, and drivers. *Figure 1-3, "Multiplexor Configurations."* illustrates the following multiplexor configurations:

- A many-to-one multiplexor routes the data from multiple upper streams to a single lower stream. Streams above the multiplexor are referred as upper streams. Streams below the multiplexor are lower streams.
- A one-to-many multiplexor routes messages from one upper stream to multiple lower streams. Streams below the driver are referred to as lower streams.
- A many-to-many multiplexor routes messages from multiple upper streams to multiple lower streams.

Figure 1-3 Multiplexor Configurations



A multiplexor is a special type of a pseudo-device driver used for multiplexing streams. With the linking facility, you can dynamically create or dismantle multiplexed streams. Multiplexor configurations can be combined to create complex variations, as required by the application.

There are two kinds of multiplexors, upper multiplexors and lower multiplexors. The upper multiplexor multiplexes the streams that are opened to the multiplexor driver by the user processes. The lower multiplexor deals with the multiple streams between the device driver and the multiplexor.

Driver developers must make explicit design choices to support lower, upper, or both types of multiplexors. STREAMS provides a flexible framework for connecting streams containing a multiplexor to other multiplexors, thereby leading to virtually unlimited interconnections. For details of multiplexing driver development, see *Chapter 4, "Modules and Drivers."*

2 STREAMS Mechanism and System Calls

This chapter describes system calls that create, use, and close a stream. The stream head processes the system calls that are made by a user level process. These system calls provide the user level facilities needed to develop applications.

This chapter describes the following:

- Creating a stream or a streams based pipe
- Writing to a stream
- Reading from a stream
- Pushing a module into a stream
- Popping a module from a stream:
- Closing a stream
- Polling a stream
- Asynchronous event notification
- Attaching and detaching a stream to a file

STREAMS System Calls

User applications use system to operate on STREAMS drivers. The following system calls are discussed in this chapter:

`open(2)`:	Creates a stream to the specified device.
`close(2)`:	Closes the stream, free the file descriptor, and if it is the last `close(2)` on this file descriptor, will dismantle the associated stream.
`read(2)`:	Enables user applications to receive messages from a stream.
`write(2)`:	Enables user applications to send messages to a stream.
`putmsg(2)`:	Enables user applications to send control information into a stream.
`putpmsg(2)`:	Sets priority control for messages written to streams.
`getmsg(2)`:	Enables user applications to receive control information from the stream.
`getpmsg(2)`:	Enables user applications to receive control information from the stream.
`ioctl(2)`:	Accesses and controls a stream. `ioctl(2)` system calls are used to perform functions that are specific to a particular device.
`poll (2)`:	Provides a synchronous mechanism for quering the stream head read queue for specific events.
`select(2)`:	Notifies the user application about specified file descriptors that are ready for reading, writing, or pending error conditions. If the specified condition is false for all of the specified file descriptors, `select(2)` blocks for the specified timeout interval. `select(2)` blocks until the specified condition is true for at least one of the specified file descriptors.
`pipe(2)`:	Connects to each other. Each stream head acts as the stream end for the other stream head. Both stream heads can be used for read and write operations.

STREAMS Library Routines

The following library routines are used in conjunction with STREAMS devices and pipes:

fattach (3C)	Attaches a specified STREAMS file descriptor or pipe to an object in the file system. The object is designated by a path. The *fattach* (3C) routine uses a file descriptor and a path name as input. *fattach* (3C) returns 0 on success and -1 on failure.
fdetach (3C)	Detaches a previously attached streams file descriptor or pipe from an object in the file system designated by a path. *fdetach* (3C) uses a path name as input. The *fdetach* (3C) routine returns 0 on success and -1 on failure.
isastream (3C)	Uses a file descriptor as input and checks to see if this descriptor is associated with a stream. The *isastream* (3C) routine returns 1 if the descriptor belongs to a streams device or STREAMS-based pipe. Else *isastream* (3C) returns 0. It returns -1 on failure.

Creating a Stream

`open(2)` Opening a STREAMS Device

A user application creates a stream by opening a STREAMS device with the *open* (2) call. This call is part of the standard UNIX I/O. It opens a UNIX file for reading, writing, or both.

Synopsis

```
#include <fcntl.h>

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

Arguments

`path`	A pointer to a path name of the STREAMS device file being opened.
`oflag`	The bitwise inclusive OR of `O_NDELAY` or `O_NONBLOCK` inclusive OR-ed with one of the read-write flags. Other flag values are not applicable to STREAMS devices and have no effect on them.
`mode`	An optional mode argument.

Read-write flags specify the mode in which the STREAMS device is to be opened. The mode can be read-only, write-only, or read-write. Determine the value for this flag by considering the type of driver and the nature of the STREAMS application accessing the driver.

Only one of the following values must be used to specify the value of `oflag`:

`O_RDONLY`	Open for read only
`O_WRONLY`	Open for write only
`O_RDWR`	Open for read-write

If none or more than one value is used, the behavior is undefined.

`O_NDELAY` and `O_NONBLOCK` are the only general flags relevant to STREAMS devices. The values of `O_NDELAY` and `O_NONBLOCK` affect the operation of STREAMS drivers and certain system calls. Refer to *read* (2), *getmsg* (2), *putmsg* (2), and *write* (2) for more information on these flags.

For drivers, the implementation of `O_NDELAY` and `O_NONBLOCK` is device specific. Each STREAMS device driver may treat these options differently. For example, while opening a STREAMS device file associated with a communication line, if either `O_NDELAY` or `O_NONBLOCK` is set, the `open()` returns without waiting for carrier. Otherwise, it does not return until the carrier is present.

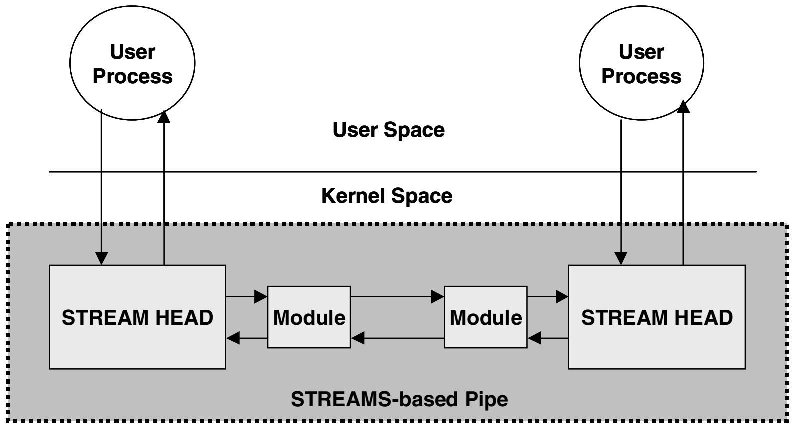
Return Values

open (2) returns an integer file descriptor upon successful completion, and a `-1` in case of failure. The corresponding `errno` is set appropriately, as with the standard UNIX File System I/O.

Creating a STREAMS-Based pipe with pipe(2)

A STREAMS-based pipe is a special case of a stream. Two streams are created and connected together when a thread executes a *pipe* (2) system call. The data flow is bi-directional in a STREAMS-based pipe. Both stream heads can be used for read and write operations. *Figure 2-1, "A STREAMS-Based Pipe,"* illustrates the bi-directional data flow in a STREAMS-based pipe.

Figure 2-1 A STREAMS-Based Pipe



Each end of the pipe maintains the status of the other end through internal data structures. Subsequent read, write, and close operations are aware of whether the other end of the pipe is open or closed.

STREAMS modules can be added to a STREAMS-based pipe with *ioctl* (2) `I_PUSH` from either end of the pipe. However, if a module is pushed onto one end of the pipe, that module can not be popped out from the other end.

STREAMS-based pipes are not attached to STREAMS-based character devices.

NOTE The standard (non-STREAMS-based) pipe is still the default on HP-UX. In order to change the default pipe to a STREAMS-based pipe, set the kernel tunable parameter `streampipes` to 1. Use `kctune` or SAM (STREAMS/UX installation will not automatically set this value). The `streampipes` is a static tunable, i.e., any modification to it will take effect after the next reboot. Refer to Appendix H, “STREAMS Kernel Tunable Parameters,” on page 309 for more details on `streampipes`.

Synopsis

```
int pipe (int fd[2]);
```

Arguments

The `pipe(2)` system call takes an array of two integers as input and returns two integer file descriptors, `fd[0]` and `fd[1]` into that integer array. These represent each end of the pipe.

Return Value

The `pipe(2)` system call returns a zero upon successful completion and -1 in case of failure. The corresponding error is available in `errno`, as with all standard UNIX file system I/O.

Writing to a Stream

User applications can write to a stream or a STREAMS-based pipe using the `write(2)`, `putmsg(2)`, or `putpmsg(2)` system call. The three function calls mirror the functionality of the corresponding message retrieval functions `read(2)`, `getmsg(2)`, and `getpmsg(2)`. These system calls are described in the *"Reading From a Stream" on page 37*.

The write(2) and writev(2) System Calls

A user application can write data to a stream using the `write(2)` and the `writev(2)` system calls.

Synopsis

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, int nbytes);

#include <sys/uio.h>
ssize_t writev(int fd, const struct iovec *iov, int iovcnt);
```

Arguments

`fd`	STREAMS device file descriptor.
`buf`	Pointer to a buffer containing the data to be written to the stream.
`nbytes`	Number of bytes to be written.
`iov`	An array of `struct iovec` (`io` vectors) that contain the data to be written.
`iovcnt`	Number of elements in the previous array.

Return Values

Upon successful completion, `write(2)` or `writev(2)` will return the number of bytes written to the stream. This number does not exceed the value specified in the `nbytes` argument. If `writev(2)` fails for any reason, -1 is returned and `errno` is set to the appropriate value.

`writev(2)`is similar to `write(2)`, except that it writes data from buffers (`io` vectors) specified by members of the `iov` array (`iov[0]`, `iov[1]`, ..., `iov[iovcnt-1]`). `iovcnt` is valid if its value is greater than 0 and less than or equal to {`IOV_MAX`}, defined in <`limits.h`>. The `iovec` structure consists of a base address (`iov_base`) and a byte length (`iov_len`) of an area in memory to be written. The `writev(2)` function always writes from one `iovec` member completely before moving on to the next. The `iovec` structure is as follows:

```
          struct iovec {
               void *iov_base; /* Starting address */
               size_t iov_len; /* Number of bytes */
          };
```

Message Priority

The `write(2)` system call can only write data messages to a stream. Data messages are always written as normal messages.

Maximum Packet Size Limit

The value of `nbytes` must fall within the acceptable STREAMS packet size range. STREAMS imposes two levels of constraints for writing messages. The maximum sizes of a STREAMS data message and a STREAMS control message are defined by `STRMSGZSZ` and `STRCTLSZ` respectively.

The minimum and maximum acceptable STREAMS packet sizes are defined by the topmost module in the stream. A STREAMS packet is a segment of a message written to the stream. The length of a packet must be less than or equal to the maximum packet size defined in the topmost module.

The following conditions apply to STREAMS packet size:

- If `nbytes` falls within the packet size range, `nbytes` will be written.
- If `nbytes` exceeds the maximum packet size, and the minimum packet size value is 0, *write* (2) will split the input buffer into segments of the maximum allowed packet size, and send them downstream. The last segment can be less than the maximum packet size.
- If `nbytes` is not within the acceptable packet size range, and the minimum packet size value is not 0, *write* (2) fails with an appropriate `errno`.
- *write* (2) enables the user application to write zero-length messages to a regular stream. The concept of zero-length messages may be relevant to some applications that want to use them as terminators and associated conditions. *write* (2) of a zero-length message to a STREAMS-based pipe writes nothing to the pipe, and still returns a 0 back to the application. The `I_SWROPT` *ioctl* (2) command overrides this default behaviour. *Appendix A, "STREAMS IOCTL Commands,"* describes the relevant *ioctl* (2) commands.

Streams Flow Control Conditions

A stream can encounter internal limits when applications attempt to write new messages to it. The following conditions should be noted for STREAMS *write* (2) calls:

- If the stream was opened without the `O_NDELAY` or `O_NONBLOCK` flag, and the buffer is either not written to the stream at all or partially written to the stream when the stream-full condition occurs, *write* (2) will block until data can be accepted.
- If the stream was opened with the `O_NDELAY` or `O_NONBLOCK` flag, and the stream-full condition occurs before any data from the buffer is written to the stream, *write* (2) will return -1, and set `errno` appropriately.
- If the stream was opened with the `O_NDELAY` or `O_NONBLOCK` flag and the buffer has been partially written when the stream-full condition occurs, *write* (2) will successfully terminate and return to the application with the actual number of bytes written to the stream.

In many respects, a *write* (2) operation to a STREAMS device (i.e., to a stream opened on a STREAMS device) is identical with *write* (2) on a file.

putmsg(2) and putpmsg(2)

The *putmsg* (2) and *putpmsg* (2) calls also write messages to streams. These system calls can handle control messages and message priority information.

Synopsis

```
#include <sys/stropts.h>
int putmsg (int fd, struct strbuf *ctlptr, struct strbuf *dataptr, int *priflag);
int putpmsg (int fd, struct strbuf *ctlptr, struct strbuf *dataptr, int *msgband, int *priflag);
```

`fd`	STREAMS file descriptor.
`ctlptr`	Pointers to `strbuf` structures containing control information.
`dataptr`	Pointers to `strbuf` structures containing data.
`strbuf`	A structure defined in `<sys/stropts.h>` to hold the control and data information to be written to the stream. The user process issuing the *putmsg* (2) or *putpmsg* (2) system calls uses the following `strbuf` structure:

```
struct strbuf {
     int maxlen;    /* maximum buffer length                 */
     int len;       /* actual length of message to be written */
     char *buf;     /* Pointer to the message buffer          */
};
```

NOTE Unlike the *getmsg* (2) and *getpmsg* (2) calls, the `maxlen` attribute is not used by *putmsg* (2).

`priflag`	Priority of the message to be written to the stream (**Ordinary** or **High**).
`msgband`	Band priority of the message to be written to the stream (applicable to ordinary messages only).

Return Values

Upon successful completion, *putmsg* (2) and *putpmsg* (2) return 0. Otherwise, they return -1 and set `errno` appropriately, as with all UNIX File system I/O.

Writing Control and Data Messages

The *putmsg* (2) and *putpmsg* (2) can write the control, data, or both the parts of the message. To write any part of the buffer, the corresponding buffer pointer (`ctlptr` or `dataptr`) must be non-null and the corresponding len field must be greater than -1. If the buffer pointer (`ctlptr` or `dataptr`) is null or the corresponding len field is -1, then nothing is written for that part of the message.

Message Priority

The priority of a message written to a stream is determined by the control part of the message. In the absence of a control part (as is the case with *write* (2)), the message priority is set to normal by default. To explicitly set the priority of the data part of a message, both the control part and the data part must be present as arguments to *putmsg* (2) and *putpmsg* (2).

For *putmsg* (2), the `priflags` argument takes one of two values — `RS_HIPRI` or 0.

- If the control part is specified and `priflags` is set to `RS_HIPRI`, a high priority message will be written to the stream.
- If no control part is specified, and `priflags` is set to `RS_HIPRI`, *putmsg* (2) fails, and sets `errno` .

- If `priflags` is set to 0, a normal message is written to the stream.
- If no control or data part are specified and `priflags` is set to 0, no message is written. However, *putmsg* (2) returns 0, indicating successful completion.

For *putpmsg* (2), the `priflags` argument is a bitwise-OR of the mutually exclusive constants, `MSG_HIPRI` and `MSG_BAND`. The `priflags` and `msgband` arguments work in conjunction to determine the message priority and band value. The following points apply to the `priflags` argument of `putmsg(2)`:

- If `priflags` is set to 0, *putpmsg* (2) fails and sets `errno`.
- If `priflags` is set to `MSG_HIPRI`, the control part of the message is present, and `msgband` is set to 0, a high priority message will be written.
- If `priflags` is set to `MSG_HIPRI`, and either no control part is specified or `msgband` is set to a non-zero value, *putpmsg* (2) fails, and sets `errno`.
- If `priflags` is set to `MSG_BAND`, and the control part of the message is present, a band message will be written. It is important to set the `msgband` argument to the desired band priority level. An incorrect band priority value causes *putpmsg* (2) to fail with an appropriate `errno`.
- If no control or data part of the message is specified, and `priflags` is set to `MSG_BAND`, no message will be written, but the function value returned by *putpmsg* (2) will be 0, indicating successful completion.

Streams Flow Control conditions

Except for the following cases, *putmsg* (2) and *putpmsg* (2) will block when the stream is not able to accept any more messages:

- High priority messages are blocked. They cause the *putmsg* (2) and *putpmsg* (2) function calls to fail.
- Non-High priority messages do not block if the `O_NONBLOCK` flag was set in the *open* (2) call for the stream. They cause *putmsg* (2) and *putpmsg* (2) to fail.

Reading From a Stream

User applications can read from a stream or a STREAMS-based pipe using the `read` (2), `readv` (2), `getmsg` (2), or `getpmsg` (2) system calls.

read(2) and readv(2)

The `read` (2) and `readv` (2) system calls are used by user applications to read message data from a stream. By default, control data will be ignored by these calls.

Synopsis

```
#include <unistd.h>
size_t read (int fd, void *buf,size_t nbytes);

#include <sys/uio.h>
size_t readv (int fd, const struct iovec *iov, int iovcnt);
```

Arguments

`fd`	STREAMS file descriptor.
`buf`	Pointer to the user buffer into which the data will be read.
`nbytes`	Number of bytes requested from the stream.
`iov`	The data from the stream read into `iov` array.
`iovcnt`	Number of elements in the `iov` array.
`iovec`	A structure consists of a base address (`iov_base`) and a byte length (`iovlen`) of an area in the memory. The `readv` (2) function always fills an `iovec` member completely before moving on to the next. The `iovec` structure is as follows:

```
struct iovec {
     void *iov_base; /* Starting address */
     size_t iov_len; /* Number of bytes  */
};
```

`readv` (2) also behaves the same way as `read` (2), except that it places the data read from the stream into buffers specified by members of the `struct iovec` array (`iov[0]`, `iov[1]`, `...iov[iovcnt-1])`. The `iovcnt` is valid if it is greater than 0 and less than or equal to `{IOV_MAX}`.

Read Modes

`The read` (2) can be executed in one of the following three modes:

- Byte-stream mode
- Message-nondiscard mode
- Message-discard mode

All streams are opened with a default of byte-stream mode. You can modify the mode using the `I_SRDOPT` `ioctl` (2) command, and verify it using the `I_GRDOPT` `ioctl` (2) command.

Byte-Stream Mode:

read (2) ignores message boundaries. It continues to retrieve data from the stream until the requested number of bytes is retrieved, there is no more data to be retrieved, or a zero-length message is encountered in the stream.

If *read* (2) terminates on finding a zero-length message, it will place that message block back on the stream to be retrieved by next *read* (2) call. read (2) then returns the control back to the calling application. However, if a zero-byte message is read as the first message on a STREAM, the message is removed from the STREAM and zero is returned, regardless of the read mode.

Message Nondiscard Mode:

The *read* (2) system call will retrieve data from the stream until the requested number of bytes have been retrieved, or a message boundary is encountered.

If bytes remain in the current message at the end of a successful *read* (2) system call, they are left in the STREAM. Those bytes will be retrieved by a subsequent *read* (2).

If a zero-byte message is read as the first message on a STREAM, the message is removed from the STREAM and zero is returned. If the zero-length message is not a first message, *read* (2) discards that message and continues to read the next message.

Message Discard Mode:

This mode differs from the message nondiscard mode in the way partially read messages are handled. If bytes remain in the current message at the end of a successful read, they are discarded. A subsequent *read* (2) call begins at the next message boundary.

Zero-length messages are handled in the same manner as in the corresponding message-nondiscard mode.

The choice of read modes depends on the desired user application functionality. Most applications need to read all the data from a stream. This makes the byte-stream mode the most commonly used option.

Control Modes

Control Normal Mode:

The default control mode. It enables *read* (2) to process data only. *read* (2) fails if a message containing a control part is encountered. This default action can be changed by placing the stream in either control data mode or control discard mode with the `I_SRDOPT` *ioctl* (2) call.

Control Data Mode:

Enables *read* (2) to convert the control part of the message to data and send it back to the calling application. This mode has limited use. Most applications that need to retrieve control information will use *getmsg* (2) or *getpmsg* (2).

Control Discard Mode:

Enables *read* (2) to discard the control part of a message and continues to read the data part. This mode is useful for applications that have to use *read* (2), and the stream head read-queue contains messages with both control and data components.

Return Values

Upon successful completion, *read* (2) and *readv* (2) return the actual number of bytes read from the stream it placed in the requested buffer. This number may be less than the number of bytes requested (`nbytes`). In case of failure, *read* (2) returns -1and sets `errno`, as with all standard UNIX File System I/O.

When attempting to read a stream (other than a STREAMS-based pipe) that supports non-blocking reads and has no data currently available, `read()` will return the following:

- If `O_NONBLOCK` is set, *read* (2) returns -1 and set `errno` to `EAGAIN`.
- If `O_NONBLOCK` is clear, *read* (2) blocks until some data becomes available.

When attempting to read from an empty pipe, *read* (2) returns the following:

- If no process has the pipe open for writing, *read* (2) returns a 0.
- If a process has the pipe open for writing and `O_NONBLOCK` is set, *read* (2) returns -1 and `errno` is set to `[EAGAIN]`.
- If `O_NDELAY` is set, *read* (2) returns a 0.
- If some process has the pipe open for writing, and `O_NDELAY` and `O_NONBLOCK` are clear, *read* (2) blocks until data is written to the file or the file is no longer open for writing.

The getmsg(2) and getpmsg(2) System Calls

The *getmsg* (2) and *getpmsg* (2) system calls read messages from a stream, and place the retrieved content separately into user-specified control and data buffers.

Synopsis

```
#include <stropts.h>

int getmsg (int fd, struct strbuf *ctlptr, struct strbuf *dataptr, int *flagsp);
int getpmsg (int fd, struct strbuf *ctlptr, struct strbuf *dataptr, int *bandp, int *flagsp );
```

Arguments

`fd`	STREAMS file descriptor.
`ctlptr`, `dataptr`	Pointers to `strbuf` structures containing the retrieved control and data information respectively.
`flagsp`	Only messages of priority specified by `flagsp` are retrieved from the stream head.
`bandp`	Only ordinary messages of band priority specified by `bandp` will be retrieved from the stream head.

The user process invoking the *getmsg* (2) or *getpmsg* (2) system call uses the following `strbuf` structure (defined in `<sys/stropts.h>`) to retrieve control and data information from the stream head:

```
struct strbuf
{
     int maxlen; /* maximum buffer length */
     int len;    /* Length of message     */
     char *buf;  /* Pointer to the buffer */
};
```

Reading Data and Control Messages

The *getmsg* (2) and *getpmsg* (2) will read control and data messages based on the following conditions:

- If a buffer pointer is null or the corresponding `maxlen` is -1, the message is not be processed, and remains on the stream head read queue. If the buffer pointer is null, the corresponding `len` field returned is set to -1.
- If `buf` is not null, `len` is greater than 0, and `len` is equal to or less than `maxlen`, `len` bytes are retrieved by *getmsg* (2), or *getpmsg* (2), and the return value of the function is set to `len`. However, if `maxlen` is less than `len` and greater than 0, only `maxlen` bytes are retrieved, and the return value of the function is set to `maxlen`. The remaining bytes are left on the stream head read queue .
- If `maxlen` is set to 0, and there is a zero length control or data message in the buffer, the zero-length part is removed from the stream head read queue.

Message Priorities and Bands

By default, *getmsg* (2) retrieves the first message from the stream head. However, a process can retrieve only high-priority messages by setting the `flagsp` argument to `RS_HIPRI`. In this case, *getmsg* (2) will only retrieve high priority messages. A `flagsp` value of 0 will cause *getmsg* (2) to read all messages from the stream.

The getmsg (2) system call enables the user applications to select between high priority and ordinary messages. It operates in the **Read** all messages and **Read High** priority messages modes. If an application needs finer control over ordinary messages, the *getpmsg* (2) call should be used.

By default, *getpmsg* (2) will also process the first available message on the stream. However, the `flagsp` for *getpmsg* (2) is a bitwise-OR of the three mutually exclusive constants `MSG_HIPRI`, `MSG_BAND`, and `MSG_ANY`.

The following list describes the combination of `flagsp` and `bandp` values that enable the user application to select different groups of messages:

- An application can retrieve only high-priority message by setting `flagsp` to `MSG_HIPRI` and `bandp` to 0. *getpmsg* (2) reads only high priority messages from the stream, and leave the rest of the messages on the stream. The `bandp` value is ignored.
- An application can retrieve messages for a particular priority band by setting `flagsp` to `MSG_BAND` and `bandp` to the desired band priority value. Band messages fall into priority bands ranging from 1 to 255. The *getpmsg* (2) system call processes the next message only if it is a band priority value equal to or greater than the specified `bandp` value, or if it is a high priority message. All other messages are left on the stream.
- If a process must retrieve the first message, set the integer pointed by `bandp` to zero and `flagsp` to `MSG_ANY`.
- If `O_NONBLOCK` is not set, `getmsg()` and `getpmsg()` does not block until a message of the type specified by `flagsp` is available at the front of the stream head read queue. If `O_NONBLOCK` is set and a message of the specified type is not present at the front of the stream head read queue, `getmsg()` and `getpmsg()` fail and set `errno` to `[EAGAIN]`.

Return Values

For *getmsg* (2) and *getpmsg* (2), the `flagsp` and `bandp` arguments serve as input and output parameters. On successful completion of the *getmsg* (2) call, `flagsp` is set to either `RS_HIPRI` or 0, depending on whether the last retrieved message was of high priority. For *getpmsg* (2), `flagsp` is set to `MSG_HIPRI`, `MSG_BAND`, or `MSG_ANY` depending on the priority of the latest message retrieved. The corresponding `bandp` argument is set to 0 for high priority and normal non-band messages, and to the corresponding band priority value for band messages.

A return value of 0 indicates that an entire message was completely and successfully retrieved. If a message is partially retrieved, `MORECTL`, `MOREDATA`, or a bitwise-OR of the two constants is returned. This indicates that more control or data information is to be retrieved for the current message.

A return value of -1 indicates failure. The `errno` value is appropriately set, as with all standard UNIX File System I/O.

See the relevant HP-UX manpages for more details on these system calls.

Pushing and Popping Modules

IOCTL Commands I_PUSH and I_POP

Modules are an optional component of a stream. A module performs intermediate operations on messages as they pass between components in a stream. Modules are added into a stream by using the `I_PUSH` *ioctl* (2) command. Modules are removed from a stream by using the `I_POP` *ioctl* (2) command.

Synopsis

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

```
int ioctl(int fd, cmd ... /*, arg */);
```

Arguments

`fd`	STREAMS File Descriptor
`cmd`	IOCTL command, e.g., `I_PUSH`, `I_POP`
`arg`	Additional argument(s), if any, required by the IOCTL command. For `I_PUSH`, this argument contains a pointer to the name of the module to be pushed on to the stream. For `I_POP`, the `arg` value must be zero.

Return Values

The *ioctl* (2) returns a 0 upon successful execution, and a -1 on failure. The `errno` values are appropriately set, as with all UNIX File System I/O.

The `I_PUSH` command pushes the module pointed by `arg` on to the top of the current stream, and below the stream head. If the stream is a pipe, the module will be placed between the stream heads of both ends of the pipe. It will then invoke the open routine for the module.

The `I_POP` command removes the module just below the stream head. If the stream is a pipe, the `I_POP` directive should be executed on the side of the pipe where the corresponding `I_PUSH` command was used.

The following example illustrates the `I_PUSH` command:

```
if (ioctl(fd, I_PUSH, “caseconvert”) < 0)
{
     /*PUSH-ing Module Dynamically */
     perror(“ioctl I_PUSH failed”);
     exit(1);
}
```

The following code sample illustrates the `I_POP` directive.

```
if (ioctl(fd,I_POP, 0 ) < 0)
{
/* POP - ping Module Dynamically */
     perror(“ioctl I_POP failed”);
     exit(2);
}
```

Closing a Stream

The last `close()` to a STREAMS file dismantles the stream as follows:

- It pops modules (if any) from the stream, allowing any messages on the write queue of the module to be drained by the module processing.
- It closes the device, allowing any messages on the write queue of the driver to be drained by the driver processing.
- It dismantles the stream by freeing any messages left on the driver's queues, and freeing the queues and stream head structures.

close(2)

If `O_NDELAY` (or `O_NONBLOCK`) is clear, close will wait up to 15 seconds for each module to drain and up to 15 seconds for the driver to drain. If `O_NDELAY` (or `O_NONBLOCK)` is set, the pop is performed immediately and the driver is closed without any delay.

Synopsis

```
#include <unistd.h>

int close(int fileds);
```

Return Value

The *close* (2) call returns a 0 upon successful completion, and a -1 in case of failure. The corresponding `errno` is appropriately set, as with all standard UNIX File System I/O.

NOTE The *close* (2) frees structures allocated by STREAMS to the different components in the stream. Modules or drivers must free any internal data structure or messages.

HP recommends that all files opened by the application, including STREAMS devices and pipes, should be explicitly closed. In the case of pipes, both file descriptors should be closed.

Polling Streams

Polling, (sometimes referred to as Synchronous Polling) is a mechanism by which a STREAMS application can query one or more file descriptors (stream head read-queues) for specific events. An event can be the arrival of specific message types, the status of a stream to accept certain message types, or the occurrence of specific error conditions.

Two variations of the polling mechanisms supported by STREAMS on HP-UX are as follows:

- *poll* (2) system call
- `/dev/poll` device

The poll(2) System Call

The *poll* (2) system call monitors I/O conditions on multiple file descriptors.

Synopsis

```
#include <sys/poll.h>
int poll(struct pollfd fds[ ], nfds_t nfds, int timeout);
```

Arguments

`fds[ ]` Array of `struct pollfd`, one per file descriptor to be polled. The `pollfd` structure is defined in `<sys/poll.h>` as follows:

```
struct pollfd
{
       int    fd;        /* file descriptor     */
       short  events;    /* requested events    */
       short  revents;   /* returned events     */
};
```

`fd`	File descriptor to be polled for one or more events
`events`	Bitwise-OR of events to be polled for a file descriptor. The application can poll different `fd`'s for a different set of events. As a result, the value of events can differ from file descriptor to file descriptor.
`revents`	Return parameter containing a bitwise-OR of events that have occurred on that file descriptor.

`nfds` Number of file descriptors to be polled.

`timeout` Number of milliseconds for which *poll* (2) waits for an event if no events are pending. A negative value indicates that *poll* (2) waits indefinitely until at least one event is detected.

Events Notified by poll(2)

STREAMS supports the following events requests in the *poll* (2) function:

`POLLIN`	A non-high priority message, This is a a regular message or a priority band message at the front of the stream head read-queue.
`POLLNORM`	Same as `POLLIN`.
`POLLPRI`	A high priority message is at the front of the stream head read queue.
`POLLOUT`	A normal or priority band message can be written to the stream without being blocked by flow control. This flag is not used for high priority messages — they can be set even when the stream is under flow control.
`POLLERR`	An error has occurred on the file descriptor. The `errno` value is appropriately set.
`POLLHUP`	Stream hangup. This stream cannot have any more messages written to it.

NOTE `POLLHUP` and `POLLOUT` are mutually exclusive.

`POLLNVAL`	The `fd` for this read queue is not a valid STREAMS file descriptor.
`POLLRDNORM`	A normal (non-priority) message is at the front of the stream head read queue.
`POLLRDBAND`	A priority band message is at the front of the stream head read queue.
`POLLWRNORM`	Same as `POLLOUT`.
`POLLWRBAND`	A priority band message can be written to the stream with no flow control restrictions.
`POLLMSG`	An `M_SIG` or `M_PCSIG` message containing the `SIGPOLL` signal has reached the front of stream head read queue.

Return Values

The *poll* (2) function examines each read queue for the occurrence of one or more of the corresponding events specified by the `events` parameter. It returns to the calling application under one of the following conditions:

- At least one of the specified events occurs on at least one read queue
- The wait period reaches the timeout value

Upon return, the *`revents`* parameter for each read-queue (corresponding to each `fd` in the `pollfd` array) contains the corresponding events encountered on that read queue.

In addition to the *`revents`* parameter, *poll* (2) returns a zero or a positive number indicating the number of file descriptors (the number of stream head read queues) that contain at least one of the requested events, within the specified timeout period. A return value of 0 indicates that none of the read-queues had any of the requested events before the timeout value was reached. A return value of -1 indicates internal failure, and causes `errno` to be set appropriately.

Important Aspects of poll(2) Functionality

- Three events (`POLLERR`, `POLLHUP`, and `POLLNVAL`) will be reported in the *`revents`* parameter if and when they are encountered. These events are reported even if they are not specifically requested in the *poll* (2) call.
- The *poll* (2) ignores all negative file descriptors. This feature is relevant from a performance standpoint. STREAMS applications that deal with multiple streams are likely to define a large array of `pollfd` structures. The large array accommodates the maximum number of streams to be opened for the life of

the application. However, during application execution, there can be a small number of open streams. The unused file descriptors inside the corresponding `pollfd` structures must be set to -1. These `pollfd` structures (with a negative `fd`) are ignored by *poll* (2). If all the `pollfd` structures passed to the *poll* (2) call have negative `fd` values, *poll* (2) returns 0 and has no other results.

- The *poll* (2) system call returns to the calling application when one of the following two conditions is met. It encounters at least one read-queue with at least one of the requested events (or a `POLLERR`, `POLLHUP`, or `POLLNVAL`), or the wait period reaches the specified timeout value in milliseconds.
- A timeout value of -1 implies no timeout, that is, *poll* (2) will wait until at least one requested event (or a `POLLERR, POLLHUP` or `POLLNVAL`) occurs on one of the read queues.
- For a timeout value of 0, *poll* (2) checks to see if any of the requested events occurred on any one of the read-queues, and returns immediately, regardless of the results.
- The *poll* (2) system call behavior is not affected by the `O_NONBLOCK` flag set on any of the specified file descriptors.

The /dev/poll Interface

HP-UX offers an alternative polling mechanism using a special device, `/dev/poll` (refer to *poll* (7) for details). This device is supported by STREAMS.

The *poll* (2) system call and the `/dev/poll` device behave differently. With *poll* (2), the application sends all relevant file descriptors and the requested events as parameters. The kernel routines associated with the stream head copy these structures into the kernel memory, and then copy back the *`revents`* parameter bitmask into every `pollfd` structure. This happens with every *poll* (2) call.

With `/dev/poll`, the user registers a set of file descriptors and a corresponding set of events. This is achieved by the application opening the (`/dev/poll`) device, and then writing the relevant `pollfd` structures and parameter values to the (`/dev/poll`) device. Next, the application polls the stream head read-queue with an *ioctl* (2) directive specifically meant for polling.

User applications working with the `/dev/poll` device use the *open* (2), *write* (2), and *ioctl* (2) functions as follows:

```
#include <sys/devpoll.h>
#include <fcntl.h>
int open("/dev/poll", O_RDWR);
int write(int fd, struct pollfd *buf, size_t nbyte);
int ioctl(int fd, DP_POLL, struct dvpoll *arg);
int ioctl(int fd, DP_ISPOLLED, struct pollfd *arg);
int close(fd);
```

The user applications performs the following steps:

1. Opens the polling device or event port (`/dev/poll`).
2. Registers file descriptors and corresponding events of interest with `/dev/poll`.
3. Deregisters (or modify selected event requests).
4. Polsl the device for the occurrence of requested events (or error conditions).
5. Closes the polling device.

Opening an event port:

Each *open* (2) call on `/dev/poll` device enables an event port where a registered set of file descriptors can be polled for relevant events. The file descriptor returned by the *open* (2) system call represents the event port as follows:

```
int evpfd;
evpfd = open("/dev/poll", O_RDWR);
```

NOTE The `/dev/poll` device can be used for *write* (2) and *ioctl* (2) operations only by the processes that opened the device. Although file descriptors are generally inherited by forked processes, the event port file descriptor behaves differently. Specifically, a child process inheriting an event port file descriptor can only perform the *close* (2) operation on it.

Registering and deregistering file descriptors on an event port:

The relevant file descriptors and corresponding events of interest are registered with `/dev/poll` using the *write* (2) call. This *write* (2) call is a standard UNIX File System write operation, and not a STREAMS *write* (2).

NOTE `/dev/poll` is not a STREAMS device.

```
int write(int evpfd, const struct pollfd *buf, size_t nbyte);
```

The *write* (2) operation automatically registers all non-negative file descriptors in the `pollfd` structure with `/dev/poll`. User applications can de-register file descriptors by setting the corresponding events attribute of that file descriptor to `POLLREMOVE`. `POLLREMOVE` is a special pseudo-event (defined for de-registration purposes) that must be the only event present in the events bitmask.

A registered file descriptor is automatically de-registered from `/dev/poll` by a *close* (2) operation.

Polling file descriptors:

User applications may poll an event port via the `DP_POLL` *ioctl* (2) command as follows:

```
int ioctl(int evpfd, DP_POLL, struct dvpoll *arg);
```

The *`arg`* parameter points to the following `dvpoll` structure:

```
struct dvpoll
{
   pollfd_t *dp_fds;         /* pollfd[ ] to be used      */
   nfds_t dp_nfds;           /* number of pollfd entries  */
   int dp_timeout;           /* milliseconds or -1         */
};
```

`dp_fds`	Pointer to an array of `pollfd` structures.
`dp_nfds`	Maximum number of `pollfd` structures in the array.
`dp_timeout`	Maximum time, in milliseconds..

The kernel copies only the affected `pollfd` structures back into `dvpoll`.

NOTE In all other respects, e.g., the setting of *`revents`* in the relevant `pollfd` structures, the return value of *ioctl* (2), the wait period behavior of timeout values, and the reporting of the 3 events (`POLLERR`, `POLLHUP` and `POLLNVAL`), the `DP_POLL` *ioctl* (2) command operates exactly like the corresponding *poll* (2) system call for the following:

- Setting *`revents`* in the relevant `pollfd` structure
- The return value of *ioctl* (2)

- The wait period behaviour of the timeout values
- Reporting the `POLLERR`, `POLLHUP`, and `POLLNVAL` events

Retrieving registered poll conditions for a file descriptor:

User applications may query `/dev/poll` for the registration status of specific file descriptors via the `DP_ISPOLLED` *ioctl* (2) command as follows:

```
struct pollfd pfd;
int ispolled;
pfd.fd = fd1;
ispolled = ioctl(evpfd, DP_ISPOLLED, &pfd);
```

If file descriptor `fd1` is registered with `/dev/poll`, the *ioctl* (2) function call will return 1, and the events attribute in the `pollfd` structure will be set to the events registered for this file descriptor.

If the file descriptor is not registered or open, *ioctl* (2) will return 0.

Closing an event port:

An event port is closed with the *close* (2) system call that specifies the event port file descriptor as follows:

```
int close(evpfd);
```

All registered file descriptors are automatically de-registered.

select(2)

Externally, the *select* (2) system call is analogous to *poll* (2), but the input parameters are structured differently. The user must specify three sets of file descriptors to test for **read**, **write**, and **exception** conditions as follows:

Synopsis

```
#include <sys/stropts.h>
#include <sys/time.h>
int select (int nfds, int *readfds, int *writefds, int *exceptfds, const struct timeval *timeout);
```

Arguments

`nfds`	Number of file descriptors to be checked.
`readfds`	Pointer to a bitmask representing a file descriptors to be checked for read-to-read
`writefds`	Pointer to a bitmask representing a set of file descriptors to be checked for ready-to-write
`exceptfds`	Pointer to a bitmask representing a set of file descriptors to be checked for exception conditions
`timeout`	Pointer to a timeout value (in microseconds) for which the *select* (2) call will block. A zero indicates no timeout, i.e., the s*elect* (2) will block indefinitely until at least one event on at least one file descriptor is detected.

Return Values

The *select* (2) returns the number of file descriptors found with an event (**read**, **write**, or **exception**). It also modifies the bitmasks to represent only those file descriptors where an event has been detected.

Differences Between select(2) and poll(2)

Unlike the `pollfd` structure that allows *poll* (2) to check a virtually unlimited number of file descriptors, the *select* (2) call expects file descriptors to be specified as three sets of bitmasks.

In HP-UX, both functions allow the user process to perform more or less than same function. As *select* (2) requires file descriptors to be specified as bitwise-ORs, it limits the number of file descriptors that can be examined in one *select* (2) call. However, the programming overhead for *select* (2) is lower than *poll* (2), where a `pollfd` structure must be initialized for each file descriptor. *poll* (2) is useful when a large number of files are to be examined and millisecond-level timeout granularity is not a constraint.

Asynchronous Event Notification

The polling method described in the previous section is synchronous. The application has to explicitly stop what it is doing, and poll each time it needs to check the occurrence of relevant events on different file descriptors.

Asynchronous Event Notification enables the user to set the event conditions in advance, and then continue with its normal operations until interrupted by the occurrence of one of those event conditions. The I_SETSIG *ioctl* (2) command enables user applications to specify one or more events for a stream.

The IOCTL Command — I_SETSIG

Synopsis

```
#include <sys/types.h>
#include <sys/stropts.h>
int ioctl(int fd, I_SETSIG, *arg);
```

I_SETSIG is the command used for setting a bit-mask of relevant events. It directs the kernel routines associated with the stream head to issue a SIGPOLL signal upon the occurrence of any of the selected events.

The *arg* parameter is a bitmask that specifies events that will signal an interrupt to the calling application. It is the bitwise-OR of any combination, except where noted, of the events listed here. For more information and a complete list of IOCTL commands, refer to *Appendix A, "STREAMS IOCTL Commands."*

S_ERROR	An M_ERROR message has reached the stream head.
S_HANGUP	An M_HANGUP message has reached the stream head.
S_HIPRI	A high priority message is present on the stream head read queue.
S_INPUT	A message other than an M_PCPROTO has arrived on a stream head This argument is maintained for compatibility with prior UNIX System V releases. This argument is set even if the message is of zero length.
S_MSG	A STREAMS signal message containing the SIGPOLL signal has reached the front of the stream head read queue.
S_OUTPUT	The write-queue just below the stream head is no longer full.
S_RDBAND	A priority band message has arrived at the stream head read queue. This is set even if the message is of zero length.
S_BANDURG	When used in conjunction with S_RDBAND, a SIGURG is generated instead of SIGPOLL when a priority message reaches the front of the stream head read queue.
S_RDNORM	An ordinary (normal) message has arrived at the stream head read queue. This is set even if the message is of zero length.
S_WRBAND	A priority band greater than zero of a queue downstream exists and is writable. This notifies the user that there is room on the queue for sending priority data downstream.
S_WRNORM	Same as S_OUTPUT.

An *arg* parameter value of zero can be used to deregister the I_SETSIG events set by a previous *ioctl* call.

Following an I_SETSIG setup, the stream head generates a SIGPOLL signal for the user application when relevant events occur. It is the responsibility of the user application to appropriately handle this signal. A typical setup for signal handling is:

```
signal (SIGPOLL, OnRecv);
```

`OnRecv` is a user-specified function to be called when the user process receives a `SIGPOLL` signal.

Refer to the *signal* (2) manpage for details on signal handling by user applications.

Attaching and Detaching a Stream to a File - Named Streams

User applications can associate a stream or STREAMS-based pipe with an existing node in the file system name space. This enables other processes to communicate with this process. For example, a process creates a pipe and names one end of the pipe using `fattach()`. When another unrelated process opens this file, it gets access to the named end of the pipe. The two processes can then communicate with each other like parent and child processes communicating over the pipe. STREAMS users can access this feature by using the `fattach()` and `fdetach()` library calls.

The `fattach()` library routine internally uses a special file system called `ffs` to support named streams.

The fattach (3C) System Call

`fattach()` attaches a STREAMS file descriptor to an object in the file system name space. A STREAMS file descriptor refers to either a STREAMS-based pipe or a STREAMS device driver.

Synopsis

```
#include <stropts.h>
int fattach(int fd, const char *path);
```

Arguments

`fd`	A STREAMS file descriptor.
`path`	Path name of an existing object in the file system.

The path cannot have a stream already attached to it. The path cannot be a mount point for a file system or the root of a file system. The caller must be an owner of the path with write permission or a user with the appropriate privileges to attach the file descriptor. A STREAMS device or pipe can be attached to more than one node in the file system name space. On attaching to a file system node, any operation on any of these paths acts on the STREAMS device or pipe, instead of the file system object path.

Once the stream is named, the `stat(2)` system call on the path shows information for the stream. If the named stream is a pipe, the `stat(2)` information shows that the path is a pipe. If the stream is a device driver or a pseudo-device driver, the path appears as a device. Following are the attributes of a fattached stream: the permissions, group ID, user ID, and times are set to those of the path. The number of links and the size as well as the device number are set to those of the STREAMS device or pipe designated by the `fd` parameter. If any attributes of the fattached stream are subsequently changed, the attributes of the underlying object are not affected. For example, `chown(2)`.

Return Values

Upon successful completion, fattach(3) returns 0. Otherwise, it returns a value of -1, and errno is set to indicate the error.

The fdetach(3C) System Call

The `fdetach()` system call detaches a STREAMS file descriptor from an object in the file system name space.

Synopsis

```
#include <stropts.h>
int fdetach(const char *path);
```

Arguments

`path`	Path name of an existing object in the file system.

The fdetach () function detaches a file descriptor from its filename in the file system. The path argument refers to the path that was previously attached using fattach (). The caller must own path or have write permission or appropriate privileges (PRIV_MOUNT) to detach a file descriptor. As a result of the fdetach () operation, the node's status and permissions are restored to the state prior to the file attaching to the node. Subsequent operations on the path will affect only the file system node and not the attached file.

If one end of a pipe is named, the last close of the other end causes the named end to be automatically detached. If the named stream is a device and not a pipe, the last close does not cause the stream to be detached. In this scenario, fdetach (3)/fdetach (1M) has to be explicitly issued to detach the named STREAMS device.

Return Values

Upon successful completion, the fdetach(3C) function returns a value of 0. Otherwise, it returns a value of -1, and `errno` is set to indicate the error.

The isastream(3C) System Call

`isastream()` tests if a file descriptor refers to a STREAMS device or a STREAMS-based pipe.

Synopsis

```
#include <stropts.h>
int isastream(int fd);
```

Arguments

`fd`	An open file descriptor.

The isastream(3C) determines whether an open file descriptor fd corresponds

to a STREAMS device or STREAMS-based pipe.

Return Values

Upon successful completion, the `isastream()` function returns a value of 0 when the file descriptor of the open file specified by `fd` is a STREAMS device or STREAMS-based pipe, and if it is not a stream, but is a valid open file descriptor. Otherwise, a value of -1 is returned, and `errno` is set to indicate the error.

File Descriptor Passing

Named streams are useful for passing file descriptors between unrelated processes. A user process can send a file descriptor to another process by invoking the ioctl(2) I_SENDFD on one end of a named stream. This sends a message containing a file descriptor to the stream head at the other end of the pipe. Another process can retrieve that message containing the file descriptor by invoking the ioctl(2) I_RECVFD on the other end of the pipe.

See Appendix - A for more information on `I_SENDFD` and `I_RECVFD` `ioctl` commands.

3 Messages

Messages are the means of communication within a stream. The objects passed between the stream components are pointers to messages. This chapter covers the structure, processing, and flow control of STREAMS messages.

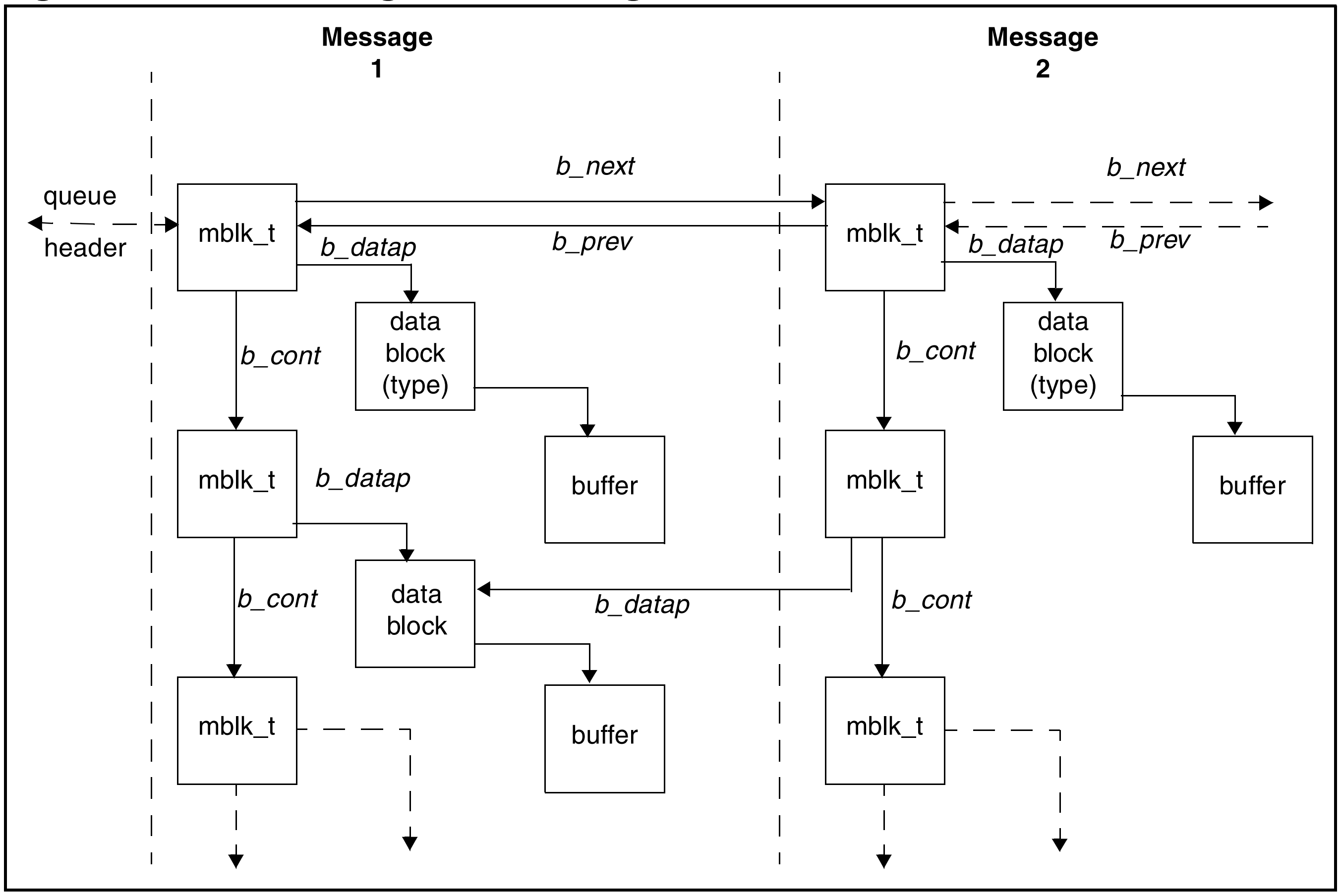
This chapter addresses the following topics:

- Message Structures
- Message Processing and Flow Control

Message Structures

All STREAMS messages consists of one or more message blocks as shown in *Figure 3-1, “A Message and Its Linkage.”* STREAMS message is comprised of two data structures — a **message block** and a **data block**. The message block represented by `msgb` describes the message, the data block represented by `datab` describes the data and contains the pointer to the variable-length data buffer.

Figure 3-1 A Message and Its Linkage



The message block and data block are represented and defined as `struct msgb` (`mblk_t`) and `struct datab` (`dblk_t`) respectively in `<sys/stream.h>`.

The `mblk_t` structure has the following fields:

```
struct msgb   * b_next;         /* next message on queue */
struct msgb   * b_prev;         /* previous message on queue */
struct msgb   * b_cont;         /* next message block of message */
unsigned char * b_rptr;         /* first unread data byte in buffer */
unsigned char * b_wptr;         /* first unwritten data byte */
struct datab  * b_datap;        /* data block */
unsigned char   b_band;         /* message priority */
unsigned short  b_flag;         /* message flags restricted for STREAMS framework use only  */
```

The fields perform the following functions:

b_next and b_prev	Link the messages on a module or driver Queue's message queue.
`b_cont`	Builds a complex message from two or more message blocks.
`b_rptr` and `b_wptr`	Locate the data present in the data buffer.
`b_band`	Contains the priority band for a message.
`b_datap`	Points to the data block header.
`b_flag`	Contains a bitmask of flags interpreted by stream head.

Using Message Block Fields

- The `b_next`, `b_prev`, `b_cont,` `b_rptr`, `b_wptr` and `b_band` can be modified by drivers or modules.
- The `b_datap` must not be modified by drivers and modules.
- The following bitmask in `b_flag` field can be set or cleared by modulesor drivers.

MSGMARK	Provides a mechanism for driver or modules to mark the message.
MSGLASTCLOSE	This flag is set in the `M_CLOSE_REPL` message by the driver upon receiving `M_CLOSE` message to cause STREAMS to dismantle the stream. This is applicable only if the device associated with the stream being closed has the flag `C_ALLCLOSES` set in `d_flags` field of the `cdevsw` table corresponding to the driver being closed.

The `datab` structure has the following fields:

```
unsigned char * db_base;      /* first byte of buffer */
unsigned char * db_lim;       /* last byte+1 of buffer */
unsigned char   db_ref;       /* number of messages pointing to us */
unsigned char   db_type;      /* message type */
```

The `db_base` and `db_lim` fields point to the beginning and end (+1) of the buffer.

The `db_ref` represents the number of message blocks sharing the data block. *Figure 3-1, "A Message and Its Linkage,"* shows `Message 1` and `Message 2` sharing the data block. Multiple messages can point to the same data block to conserve the memory and avoid copying overhead. The drivers or modules can use the *dupb* () command utility to share the data block between the multiple messages. If the value of `db_ref` is greater than 1, the modules and driver must not modify the data buffer.

The `db_type` contains the message type associated with the message. The message type indicates the type of operation and represents the implicit priority associated with the message.

Drivers and modules must not modify the contents of `dblk_t`.

Message Types

Every message in STREAMS that is generated by a user application or STREAMS component has a message type attribute associated with it. The message type specifies the implicit message priority. The different types of messages are supported by STREAMS/UX and are defined in `<sys/stream.h>`, refer to *Appendix C, "Message Types," on page 251*.

STREAMS/UX supports the following message types:

Ordinary Messages

Ordinary messages are always put at the end of the message queue. These messages have the priority band (b_band) set to zero. However, any ordinary messages can be changed to priority band messages (also known as Expedited data) by modifying b_band value. STREAMS supports priority band value up to 255.

M_DATA	User data.
M_PROTO	Protocol control information.
M_IOCTL	User IOCTL request generated by stream head.
M_PASSFP	Passes file descriptor between the processes.
M_SIG	Signal sent by a driver/module to a process.
M_BREAK	Requests a driver to send a break to the hardware device.
M_DELAY	Request a driver to generate a delay on output.
M_CTL	Used for inter module communication and is implementation dependent.
M_SETOPTS	Sets stream head characterstics.
M_TRAIL	Marks the end of message after M_HANGUP.
M_RSE	Reserved for STREAMS internal use.

High Priority Messages

High priority messages are placed at the head of the message queue and are not governed by STREAMS flow control.

M_PCPROTO	Protocol control information.
M_READ	Read notification sent downstream by stream head.
M_PCSIG	Signal sent by a driver or module to a process.
M_COPYIN	Copy in user data for transparent IOCTL.
M_COPYOUT	Copy out kernel data for transparent IOCTL.
M_IOCDATA	Sent downstream by stream head in response to M_COPYIN or M_COPYOUT.
M_IOCACK	Positive acknowledge of an IOCTL request.
M_IOCNAK	Negative acknowledge of an IOCTL request.
M_ERROR	Report downstream error condition and to mark stream in error.
M_FLUSH	Discard messages in the stream.
M_HANGUP	Sent upstream by the driver to indicate that it is disconnected from the device.
M_START	Request devices to start output.
M_STARTI	Restart devices input.
M_STOP	Stop device output.
M_STOPI	Stop device input.
M_PCRSE	Reserved for STREAMS internal use.
M_CLOSE	Notify the driver of close(2) when allcloses is in effect.
M_CLOSE_REPL	Driver reply to M_CLOSE; MSGLASTCLOSE flag set and indication to dismantle the stream.

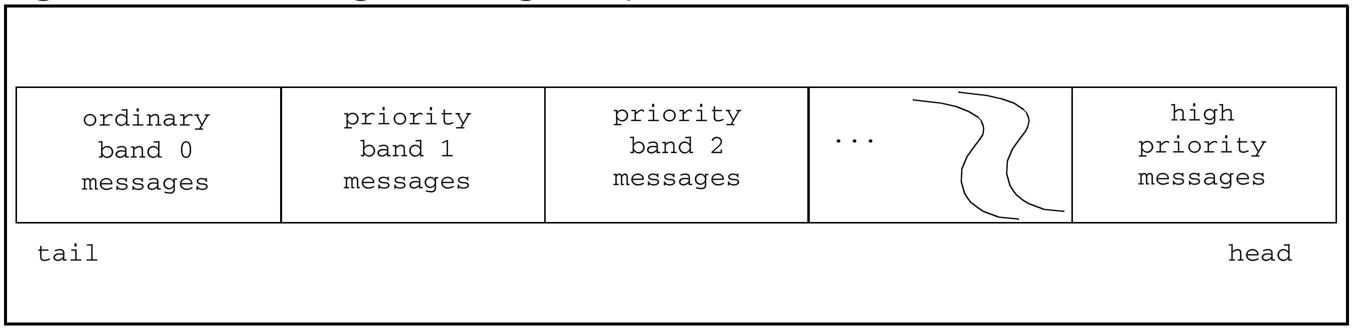
Message Queues

Message queues are essentially a linked list of messages waiting to be processed by the service procedure. The STREAMS scheduler accesses these lists through the pointers available to the head and tail of the lists in the module or driver queues as explained.

A queue will generally not use its message queue if there is no service procedure associated with it. Message queues grow as the processing of the messages on it are delayed due to a STREAMS scheduler being delayed, or when the next module is flow-controlled. The priority of the message indicates the order in which it is enqueued.

High priority messages are placed at the head of the message queue. Then, the priority band messages and finally, band zero or ordinary messages as shown in *Figure 3-2, "Message Ordering in a Queue."* The service procedure processes the enqueued messages in a **first-in first-out** (FIFO) manner.

Figure 3-2 Message Ordering in a Queue



Queue

In addition to the pointers to the message queues, the message queue also contains the entry points for message processing, flow control parameters, pointer to the next queue in the stream and so on.

Message queues are always allocated in pairs (read and write). One pair is allocated for each component of the stream. A queue-pair is allocated for each streams component, and is initialized when the streams component becomes a part of the stream. The queue pairs for the stream head and driver are allocated when the driver is opened and deallocated on closing the driver. The queue pairs for a module are allocated upon pushing the module on to the stream and deallocated on popping or removing it from the stream.

The streamtab Structure

Every STREAMS module and driver installed on a system is associated with a `struct streamtab` structure defined in `<sys/stream.h>` in addition to other structures. The information provided by the module or driver writer in this structure is used by STREAMS to initialize the queue-pair associated with the module or driver.

The `struct streamtab` contains pointers to message handling procedures and other message processing parameters defining the behavior of the module.

```
struct qinit    * st_rdinit;    /* defines read QUEUE */
struct qinit    * st_wrinit;    /* defines write QUEUE */
struct qinit    * st_muxrinit;  /* for multiplexing drivers only */
struct qinit    * st_muxwinit;  /* for multiplexing drivers only */
```

The `st_muxrinit` and `st_muxwinit` structures point to the `qinit` structure of the lower read-side and write-side of a multiplexing driver.

NOTE For all modules and non-multiplexing drivers these fields should be set to NULL.

The queue Structure

The `queue` structure defined in `<sys/stream.h>` as `queue_t` is the central building block of the queues in STREAMS. The `struct queue` has the following fields:

```
struct qinit *  q_qinfo;    /* procedures and limits for queue */
struct msgb *   q_first;    /* head of message queue */
struct msgb *   q_last;     /* tail of message queue */
struct queue *  q_next;     /* next QUEUE in Stream */
struct queue *  q_link;     /* link to scheduling queue */
void *          q_ptr;      /* to private data structure */
ulong           q_count;    /* weighted count of characters on q */
ulong           q_flag;     /* QUEUE state */
long            q_minpsz;   /* min packet size accepted */
long            q_maxpsz;   /* max packet size accepted */
ulong           q_hiwat;    /* high water mark, for flow control */
ulong           q_lowat;    /* low water mark */
struct qband *  q_bandp;    /* band information */
unsigned char   q_nband;    /* number of bands */
struct queue *  q_other;    /* pointer to other Q in queue pair */
```

On a queue-pair allocation, the contents of this structure are initialized to zero. STREAMS however initializes the following fields if specified by the modules:

- `q_qinfo` — set to the value specified in the `streamtab`
- `q_minpsz`, `q_maxpsz`, `q_hiwat` and `q_lowat` — set to the value specified in the `struct module_info`.

The module or driver open routine can optionally set the `q_ptr` to point to a private structure owned and managed by the modules/drivers.

The `q_first` and `q_last` fields refer to the first and last messages on the message queue.

The `q_next` field points to the queue associated with the next component, downstream for a write queue and upstream for a read queue.

The `q_count` field contains the total number of message bytes on the queue inclusive of all normal, banded and high priority messages. When `q_count` equals or exceeds the queue's high watermark (`q_hiwat`), STREAMS marks the queue as full.

The `q_flag` field contains a bitmask indicating the state of the queue. It can have one or more of the following values listed:

`QREADR`	Defines a read queue.
`QNOENB`	Do not enable the queue when data is placed on it.
`QENAB`	The queue is enabled to run the service procedure.
`QFULL`	The queue is full.
`QWANTR`	Enable the queue when data is placed in it.
`QWANTW`	Back enable the stream when the queue drains.
`QUSE`	The queue is allocated and ready for use.
`QBACK`	The queue is back-enabled.
`QOLD`	The queue supports module or driver interface to open or close developed prior to UNIX System V Release 4.0.
`QWELDED` —	The queues are welded.

The `q_nband` field indicates the number of priority bands present in the queue. The `q_bandp` points to the linked lists of these priority band queues.

Using Queue Information

Not all the fields in the queue structure can directly be manipulated or accessed by the modules or drivers.

- The `q_ptr` field, is the only field which can directly be manipulated by drivers and modules.
- The `q_minpsz`, `q_maxpsz`, `q_hiwat` and `q_lowat` fields can be modified by the modules and drivers, but should be done so using the STREAMS utilities like *strqget* ()/*strqset* ().
- The `q_first` and `q_last` fields pointing to the head and the tail of the message queue, `q_count` and `q_flag` can get altered through the STREAMS utilities. The modules or drivers should not modify them directly.
- The `q_qinfo`, `q_next`, `q_other`, `q_bandp` and `q_nband` fields can be read by the modules and drivers but not changed.
- The `q_link` field is for STREAMS internal use only.

The qinit Structure

The `qinit` structure contains the addresses of the functions associated with the queue, associated module information, and an optional statistics if present. The `streamtab` structure as shown earlier for a module or driver has pointers to the `qinit` structure both for the upstream and downstream processing. The `struct qinit` is defined in `<sys/stream.h>` and contains the following fields:

```
int    *qi_putp (queue_t *, mblk_t *);    /* put procedure */

int    *qi_srvp (queue_t *);              /* service procedure */

int    *qi_qopen ();                      /* open routine */

int    *qi_qclose ();                     /* close routine */

int    *qi_qadmin (void);                 /* admin procedure */

struct module_info * qi_minfo;            /* module parameters */

struct module_stat * qi_mstat;            /* module statistics */
```

qband Structure

The `qband` structure maintains information about each priority band in a queue. These `qband` structures are not pre-allocated per queue. They are allocated when a message with a priority band value greater than zero is placed on a queue. The maximum number of `qband` structures in a queue is 255. Once allocated, the `qband` structure remains associated with the queue until the queue gets freed. The `struct qband` is defined in `<sys/stream.h>` and contains the following fields:

```
struct qband *  qb_next;        /* next band for this queue */

ulong           qb_count;       /* weighted character count in band */

struct msgb *   qb_first;       /* head of message queue */

struct msgb *   qb_last;        /* tail of message queue */

ulong           qb_hiwat;       /* high water mark */

ulong           qb_lowat;       /* low water mark */

ulong           qb_flag;        /* state */
```

The `qb_next` field points to the next priority band in the queue. The `qb_count` field contains the total number of bytes for all the messages in the band. The `qb_first` and `qb_last` fields point to the first and the last message in the band. Each band also contains a separate high and low water mark, `qb_hiwat` and `qb_lowat`. The `qb_flag` flag denotes the state of the band, represented by the following three values:

`QB_WANTW`	Back enable the stream when this queue or band drains.
`QB_BACK`	The queue is back enabled.
`QB_FULL`	The queue is full.

Message Processing and Flow Control

Message Processing

Each queue in a stream is associated with a `put` procedure and an optional `service` procedure. The only exception is the driver's read-queue, which may not have a `put` procedure. The `put` procedure and the `service` procedure are the two routines to process the messages.

The `put` procedure is used to process messages immediately. It processes the message as required when it receives the message from the previous queue. Depending on the nature of the message and the availability of the next queue in the stream, the `put` procedure can consume this message, pass it to the next component's queue for further processing, or place the messages on its message queue for deferred processing.

The `service` procedure is for deferred processing of the messages in the message queue. The `service` procedure must use the *getq* () utility to remove the message from the message queue and process the message accordingly. STREAMS guarantees that only one instance of the `service` procedure for a specific queue may run at a time. Details of the rules for `put` and `service` procedures are described in *Chapter 4, "Modules and Drivers."*

The `service` procedure is usually scheduled to run when a message is first placed into the message queue. The `service` procedure can also be scheduled by STREAMS as described in the *See "Flow Control" on page 63* section in this chapter. In addition, STREAMS provides the qenable () utility to enable the module or driver to schedule the `service` procedure directly.

STREAMS provides a set of service schedulers, defined by the `NSTRSCHED` tunable, to run the scheduled `service` procedures. By default, the number of service schedulers is equal to the number of CPUs.

Flow Control

STREAMS provides a flow control mechanism that enables module and driver developers to manage the maximum amount of data-flow in the stream. Implementing flow control is optional, but highly recommended.

To implement the flow control, the module or driver must meet the following requirements:

- Provide `service` procedure
- Set high and low water marks
- Use the `canput` () family calls to check for flow control condition

Service Procedure

A queue must contain a `service` procedure to implement the flow control. The `service` procedure is pointed to by the `qi_srv` pointer in the `qinit` structure. A NULL `qi_srv` pointer indicates that this queue does not implement flow control.

Watermarks

Each queue is associated with the following watermarks and counter to manage the amount of data accumulated in the message queue:

q_hiwat	High watermark
q_lowat	Low watermark
q_count	Counter

When a message is placed onto the message queue, STREAMS increments `q_count` by the size of the message. When a message is removed from the message queue, STREAMS decrements the `q_count` accordingly.

The high watermark defines the maximum amount of data in bytes that can be accumulated in the message queue. Once the accumulated data in the message queue reaches the high watermark, STREAMS sets the `QFULL` flag to indicate that this message queue is full. When the message queue is full, the queue is under flow control.

The high watermark is a soft limit. Hence, STREAMS allows the accumulated data to grow above the high watermark if the preceding queue ignores the flow control condition. The low watermark defines the point where STREAMS resets the queue full condition. That is, when the accumulated data in the message queue reaches the low watermark, STREAMS resets the `QFULL` flag.

Set the high and low watermarks as follows:

1. Specify the high and low watermarks in the `mi_hiwat` and `mi_lowat` fields of the `module_info` structure.

   These values are set independently for the read queue and the write queue. Once an instance of the module is allocated, the `q_hiwat` and the `q_lowat` are initialized to the values specified in `mi_hiwat` and `mi_lowat` fields.

   When `q_band` is allocated for the band messages, the corresponding `qb_hiwat` and `qb_lowat` for each band are defaulted to equal to the value of `q_hiwat` and `q_lowat`.

2. Dynamically change the high and low watermarks at run time by using the *`strqset`* () utility.

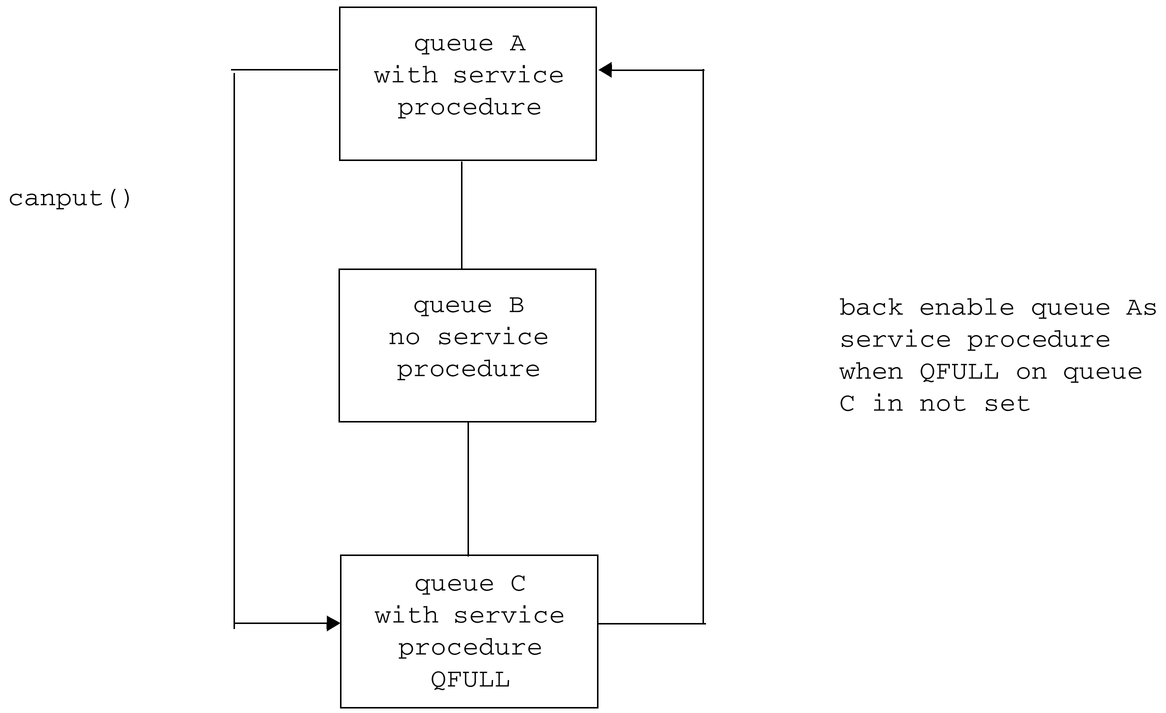
Checking Data Flow Condition

STREAMS provides the *canput* (), *bcanput* (), *canputnext* () and *bcanputnext* () utilities to check the flow control condition. Their definition are available in the *"Message Processing Utilities"* section.

The module implementing flow control usually checks the flow control condition before it passes the message to its next queue. HP recommends that high priority messages are not flow controlled. However, if high priority messages are placed onto the message queue, they will be read and processed before other messages in the queue.

Implementing flow control is optional. For example, *Figure 3-3, "Flow Control,"* shows that queue A and queue C implement flow control, queue B is not flow controlled.

Figure 3-3 Flow Control


queue A with service procedure
canput()
queue B no service procedure
back enable queue As service procedure when QFULL on queue C in not set
queue C with service procedure QFULL

When queue A uses `canput` () to check for available space, `canput` () searches through the stream. `canput` () starts searching from queue B, until it finds queue C that contains a `service` procedure. Then, `canput` () checks if `QFULL` is set on queue C. If `QFULL` is not set, `canput` () returns 1. Else, `canput` () marks `QWANTW` on queue C, and returns 0.

Back-Enabling Mechanism

STREAMS provides a back-enabling mechanism to schedule the `service` procedure to processing the messages in the message queue. This mechanism works as follows:

When the message queue reaches the high water mark, STREAMS marks the message queue as `QFULL`. Once *canput* () detects the `QFULL` condition, *canput* () sets the `QWANTW` flag.

When the accumulated data drops from the high water mark to the low watermark, STREAMS clears the `QFULL` flag, and checks if `QWANTW` is set. If `QWANTW` is set, STREAMS schedules the `service` procedure of the nearest preceding queue to process the messages.

For example, in *Figure 3-3, “Flow Control,”* queue C is in the `QFULL` state. A *canput* () from queue A sets `QWANTW` on queue C, and returns 0. Upon return from `canput` (), queue A may place the message onto the message queue for processing when queue C becomes available at a later time.

When queue C’s state changes from `QFULL` to `non QFULL`, STREAMS checks if `QWANTW` is set on queue C by any previous `canput` (). If `QWANTW` is set, STREAMS schedules the service procedure of queue A to process the messages blocked due to the queue full condition encountered by previous `canput` ().

The details of how STREAMS implements the back-enabling mechanism is transparent to the stream module and drivers. As long as the module or driver meets the three requirements listed in the *“Flow Control”* section, STREAMS ensures the module or driver’s `service` procedure is scheduled when the `QFULL` state is unset.

Common STREAMS Utilities

This section provides an overview of the most common utilities used by modules or drivers. *Appendix B, "STREAMS Utilities Supported by HP-UX,"* contains the complete set of STREAMS utilities relevant to STREAMS programmers. All these utilities are exported via `<sys/stream.h>`.

Message Allocation and Deallocation

```
mblk_t * allocb(int size, unsigned int pri);
```

allocb () Allocates a message block header, data block header and the data buffer of length determined by size (in bytes). When successful, it returns a pointer to the newly allocated message block of type `M_DATA`. The `b_band` field is set to zero. Modules and drivers can set this field, if needed. The `pri` is an unused parameter and is maintained only for compatibility with the applications developed prior to UNIX System V Release 4.0.

The `allocb` () utility returns a NULL pointer if it fails to allocate the requested memory. Module and Drivers need to take care of this condition where *allocb* () fails to allocate the requested memory.

```
int bufcall(unsigned int size, int pri, bufcall_fcn_t func, bufcall_arg_t arg);
```

bufcall () Serves as a memory availability notification mechanism. It can be used if `allocb` () fails to allocate memory. If the *allocb* () returns NULL, modules and drivers may invoke `bufcall` () to recover from the `allocb` () failure. This utility invokes the callback function when a buffer of size bytes is available. The callback function must not sleep and cannot reference `u_area`. The `pri` is not used.

The `bufcall` () utlity returns 1 on success, and 0 on failure. On a failure return, `func` is never invoked and this indicates the system is temporarily unable to allocate the required internal memory.

```
mblk_t *esballoc(uchar_t *base, int size, int pri, frtn_t *fr_rtn);
```

esballoc () Allocates a new message block and data block header. The user-supplied buffer pointed at by the base is used as data buffer. The `b_rptr`, `b_wptr`, `db_base` and `db_lim` are set appropriately, based on base and size parameters.

The `free_func` () pointed by `fr_rtn` is called with `free_arg` by `freeb` () when `db_ref` count is 1.

```
void freeb(mblk_t *bp);
```

freeb () Deallocates the message block descriptor `bp`. The data block associated with `bp` is not freed if the `db_ref` count is greater than 1. If the `db_ref` is greater than 1, `freeb` () will free only the message block header and decrements the `db_ref` count.

```
void freemsg(mblk_t *bp);
```

freemsg () Will free the message pointed at by `bp`. This utility internally uses `freeb` () to free the individual message blocks by traversing the `b_cont` list.

```
mblk_t *copyb(mblk_t *bp);
```

copyb () Allocates a new message block and updates all the information pointed at by `bp`. All the data between the `b_rptr` and `b_wptr` pointers of a message block are copied to the new message block.

The `copyb` () returns a pointer to the newly allocated message block for successful return. In case of failure, it returns a NULL pointer.

```
mblk_t *copymsg(mblk_t *mp);
```

copymsg () Uses *copyb* () to copy the message blocks contained in the message pointed by `bp` to a newly allocated message block, and links the new message blocks to form the new message.

```
mblk_t *dupb(mblk_t *bp);
```

dupb () Takes a pointer to a message block descriptor (`mblk_t`) as an input parameter, allocates a new message and initialized it by copying the contents of the input message block. The new message block descriptor points to the same data block as the input message block descriptor. The `db_ref` in the data block (`dblk_t`) is incremented by 1.

When `db_ref` is greater than 1, modules and drivers must be aware that any changes made to the data buffer affects all the message blocks sharing this data block. If the contents of a message block with a reference count greater than 1 is to be modified, *copymsg* () should be used to create a new message and only the new message block should be modified.

Upon success, *dupb* () returns a pointer to the new message block. Otherwise it returns a NULL pointer.

```
mblk_t *dupmsg(mblk_t *bp);
```

dupmsg () Copies all the individual message block descriptors pointed by `bp`, and then using `b_cont` pointer links the new message blocks to form the new message. Internally this utility uses *dupb* (). The *dupmsg* () does not copy data buffers.

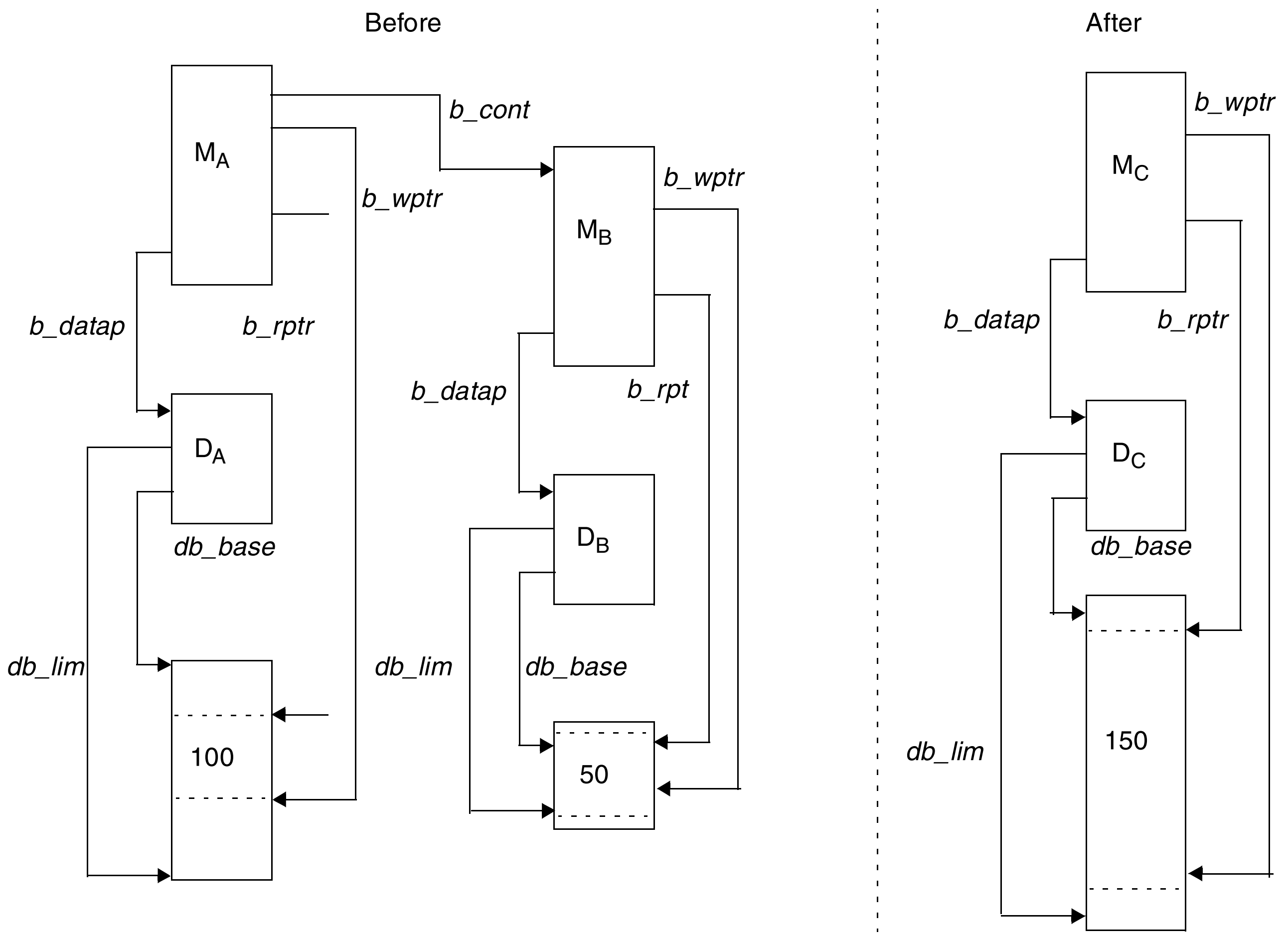
Upon success, *dupmsg* () returns a pointer to the new message. In case of failure, it returns a NULL pointer.

```
int pullupmsg(mblk_t *mp, int len);
```

pullupmsg () Concatenates and aligns the number of bytes as represented by `len` bytes stored in a complex message `mp`. The `pullupmsg` only concatenates messages of the same type. See the *Figure 3-4, "Pulling Up a Complex Message,"* to see the results of the *pullupmsg* () operation. If len is -1 then *pullupmsg* () will concatenate and align the entire contents of all the messages into a single data block.

The *pullupmsg* () returns 1 on success and 0 in case of failure.

Figure 3-4 Pulling Up a Complex Message



Message Processing Utilities

```
int putnext(queue_t *q, mblk_t *mp)
```

putnext () Is used for passing messages to the next queue in a stream. This utility calls the `put` procedure associated with the next queue in a stream and passes it a message block pointer as an argument.

```
int putq(queue_t *q, mblk_t *mp)
```

putq () Is typically used by the `put` procedure for deferred processing of the messages. The *putq* () utility puts the message `mp` in the message queue based on its priority. The service procedure is enabled if a high priority message is put in the queue and `QNOENAB` is unset if flow control parameters are updated for this queue.

The *putq* () utility returns 1 on success. Otherwise it returns 0.

```
int insq(queue_t *q, mblk_t *emp, mblk_t *nmp)
```

insq () Inserts a driver supplied new message block `nmp` into a specific place in the message queue, right before the message `emp` already in the queue. If `emp` is NULL, place the `nmp` at the end of the queue.

The priority band of the new message must adhere to the following ordering:

```
emp->b_band <= nmp->b_band <= emp->b_prev->b_band
```

The flow control parameters for the `q` specified are updated.

insq () Returns 1 on success. Otherwise it returns 0.

```
mblk_t *getq(queue_t *q)
```

getq () Returns the next available message from the message queue associated with the `q`. This is typically used by the `service` procedure to process the messages. If there are no messages, this routine will return a NULL pointer and `QWANTR` is set so when the next message is placed, the `service` procedure will be scheduled. It updates the flow control parameters.

```
int putbq(queue_t *q, mblk_t *mp)
```

putbq () Returns a message `mp` back to the beginning of the queue pointed at by `q`. The position of the `mp` depends on the message type. The flow control parameters are updated and the `service` procedure is scheduled in accordance with the same set of rules mentioned in *putq* ().

A `service` procedure must not use *putbq* () for high priority messages.

The *putbq* () utility returns 1 on success and 0 on failure..

```
void flushq(queue_t *q, int flag)
```

flushq () Removes the message from the message queue associated with the `q`. If the flag is set to `FLUSHDATA`, *flushq* () frees all the `M_DATA`, `M_PROTO`, `M_PCPROTO`, and `M_DELAY` messages. `FLUSHALL` will result in flushing all messages in the message queue. The flow control parameters are updated for this queue and the nearest `service` procedure is enabled if `QWANTW` is set.

```
void flushband(queue_t *q, int pri, int flag)
```

flushband () Will free the message associated with a given priority band as specified in the `pri` parameter. If the `pri` is set to 0, then only the normal messages are freed according to the value of flag. Otherwise, messages are flushed from the band specified by `pri`. The `FLUSHDATA` & `FLUSHALL` are the valid bitmasks for the flag parameter.

If the value of `pri` is greater than `q_nband` then *flushband* () routine simply returns to the user without flushing any messages on the queue.

The flow control parameters are updated corresponding to this queue and the nearest service procedure is enabled if `QWANTW` is set.

```
void qenable(queue_t *q)
```

qenable () Provides a mechanism for drivers/modules to enable `service` procedure to run. It inserts the `q` into the linked list of queues that are ready to be invoked by the STREAMS scheduler.

```
int canput(queue_t *q)
```

canput () Checks to see if additional messages can be enqu1230.ued onto the message queue correspond to `q`. It returns 1 if queue is not full, otherwise it returns 0.

```
int bcanput(queue_t *q, int pri)
```

bcanput () Checks the flow control for the given priority band. It returns 1 if a message of priority `pri` can be placed on the message queue, otherwise it returns 0.

4 Modules and Drivers

This chapter describes the various data structures essential to modules and drivers. It also provides design guidelines for developing modules and drivers. Appropriate code examples have been provided.

This chapter addresses the following topics:

- Overview
- Data Structures
- Entry Points
- Flush Handling
- Design Guidelines
- STREAMS Module
- STREAMS Driver
- DLKM STREAMS

Overview

Modules and drivers are the processing elements on a stream, below the stream head. STREAMS modules and drivers are structurally similar. The `streamtab`, `qinit`, `module_info`, and optionally `module_stat`, structures must be declared for STREAMS drivers and modules. A STREAMS driver is a required element but modules are optional.

A STREAMS device driver is similar to a regular UNIX character device driver. It is opened as a regular driver even if it is a pseudo-driver.

The following are the key differences between a module and driver:

- A STREAMS driver, if associated with a hardware device, must be able to handle the interrupts from the device and include an interrupt handling routine. A STREAMS module does not contain an interrupt handling routine.
- A STREAMS multiplexing driver can have multiple streams connected to it. Multiplexing is not applicable to STREAMS modules.
- A STREAMS driver is initialized or deinitialized when it is opened or closed. A STREAMS module is initialized/deinitialized when it is pushed onto a stream or popped from a stream (PUSH and POP operations result in calling the open and close routines of the module).
- A STREAMS driver must send a negative acknowledgement if it does not understand the IOCTL command passed to it. A STREAMS module must pass the IOCTL to the next component in the stream.
- Like all UNIX device drivers, a STREAMS driver is also associated with a major device number in the structure `drv_info` and a minor device number. The structure `drv_info` is not applicable for a STREAMS module, and no major or minor number is associated with it.

Cloning

STREAMS provides a feature called **cloning**. Using cloning, applications opening the STREAMS driver do not need to search for an unused minor device. In HP-UX5644444664666666666666666664, cloning is implemented with the help of the clone driver with `72` as its major number.

If a STREAMS driver is implemented as a clonable driver in HP-UX, a single node in the file system may be opened to access any unused device that the driver controls. This node must be created with its major number as `72` (the major number of the clone driver). The minor number is the same as the real driver’s major number. See the *“Entry Points”* and *“STREAMS Driver”* sections for more details.

Autopush

Usually, when a STREAMS driver is opened, the stream created as a result has no modules on it. Applications have to push the required modules explicitly after opening the driver. The autopush feature in STREAMS allows administrators to specify a list of modules to be automatically pushed when a device is opened by STREAMS. The command `autopush` (1M) provides the administrative interface to configure STREAMS devices for autopush.

Dynamically Loadable Kernel Modules

Dynamically Loadable Kernel Modules (DLKM) are device drivers and other kernel modules that can be added to a running system without rebooting the system or rebuilding the kernel. Unless otherwise mentioned, the STREAMS modules and drivers are statically linked into the kernel. The extra set of routines, definitions and so on, required to make a STREAMS driver or module dynamically loadable are documented in the *HP-UX Driver Development Guide*.

Data Structures

A STREAMS module or driver must define and initialize three data structures — `module_info`, `qinit`, and `streamtab`. An optional `module_stat` structure can also be defined and initialized. In addition to these data structures, HP-UX requires all modules and drivers to define `streams_info_t`. Drivers must define `drv_info_t` and `drv_ops_t`.

DLKM modules require additional data structures to be defined. These are described in the *"DLKM STREAMS"* section.

The module_info Structure

The `module_info` structure provides the information about a module or driver. The information in this structure is shared between both the read and write queues:

```
unsigned short  mi_idnum;       /* module ID number */
char *          mi_idname;      /* module name */
long            mi_minpsz;      /* min packet size use */
long            mi_maxpsz;      /* max packet size use */
ulong           mi_hiwat;       /* high water mark, for flow control */
ulong           mi_lowat;       /* low water mark, for flow control */
```

`mi_idnum`

Represents a number used to identify the module or driver. When this field is set to zero, STREAMS assigns a unique number. If it is set by the driver or module, case, STREAMS cannot ensure a unique number. The `mi_idnum` can be used by user processes and the STREAMS driver or modules in tracing events and logging errors as specified in `strlog` (7). See *Appendix B, "STREAMS Utilities Supported by HP-UX."*

`mi_idname`

Is a pointer to an array of characters representing a unique name. The name identifies the module or driver. This parameter is set by the module or driver developer. STREAMS truncates the extra characters if the name exceeds `FMNAMESZ`.

`mi_minpsz` and `mi_maxpsz`

The minimum and maximum packet size of an `M_DATA` message that are accepted by a module or driver. These values are copied onto `q_minpsz` and `q_maxpsz` during *open* (), for a driver or the *push* () for a module.

These limits are used by the stream head when sending messages downstream. The stream head uses `q_maxpsz` of the top most module to fragment the data, if required, for large writes. These are advisory limits between modules and drivers.

`mi_hiwat` and `mi_lowat`

The values represent the flow control parameters for a queue. These values represent the maximum and minimum number of bytes that can be put onto the message queues. They are copied onto `q_hiwat` and `q_lowat`, respectively.

The qinit Structure

The `qinit` structure is as follows:

```
int     (*qi_putp) __((queue_t *, mblk_t *)); /*  put procedure     */
int     (*qi_srvp) __((queue_t *));           /*  service procedure */
int     (*qi_qopen)();                        /*  open procedure    */
int     (*qi_qclose)();                       /*  close procedure   */
int     (*qi_qadmin) __((void));              /*  qadmin routine    */
struct module_info * qi_minfo;                /*  module info       */
struct module_stat * qi_mstat;                /*  module stat       */
```

The `qinit` structure defines the queue procedures; put(), service(), open(), close(), and admin(). The qinit data structure also contains pointers to module_info and an optional module_stat structure.

NOTE The `qi_mstat` functionality is not supported by the STREAMS subsystem. This should be set to NULL. Drivers or modules must not modify the contents of this structure.

Contents of the `qinit` structure can be shared between the read and write queues. However, modules or drivers must define `qi_open`, `qi_qclose`, and `qi_minfo` in the read side `qinit` structure. Each queue must define a `put` procedure in its `qinit` structure. For a driver, the read side `put` procedure can be set to NULL.

The `streamtab` structure is as follows:

```
struct qinit  * st_rdinit;   /* read side of a module/non-mux driver  */
struct qinit  * st_wrinit;   /* write side of a module/non-mux driver */
struct qinit  * st_muxrinit; /* lower read side  of a mux driver      */
struct qinit  * st_muxwinit; /* lower write side of a mux driver      */
```

For modules and non-multiplexing drivers, only the first two fields are valid and set the remaining two fields to NULL. The first two fields represent the `qinit` structures of the read and write queues.

For multiplexing drivers, the first two fields contain the address of the read-side or write-side `qinit` structures of an upper multiplexor. The next two entries contain the address of the read-side or write-side `qinit` structures of a lower multiplexor.

The `streamtab` structure must be a globally accessible and should not be defined with a static type qualifier. The modules or driver must not modify the contents of this structure at run time.

The module_stat Structure

The `module_stat` structure is optional and it can be used to hold statistics about a driver or module. The STREAMS subsystem does not provide services to set or access the information in the `module_stat` structure as follows:

```
long    ms_pcnt;        /* count of calls to put proc */
long    ms_scnt;        /* count of calls to service proc */
long    ms_ocnt;        /* count of calls to open proc */
long    ms_ccnt;        /* count of calls to close proc */
long    ms_acnt;        /* count of calls to admin proc */
char    * ms_xptr;      /* pointer to private statistics */
short   ms_xsize;       /* length of private statistics buffer */
```

The following code snippet illustrates the declaration of mandatory structures for a dummy STREAMS driver named `drv`:

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

#include <sys/param.h>

#include <sys/stream.h>

#include <sys/strenv.h>


static int drv_open(), drv_close(), drv_put(), drv_srv();


static struct module_info drv_minfo = {

       0, "drv", 0, 512, 65535, 8192

};


static struct qinit drv_rinit = {

       drv_put, drv_srv, drv_open, drv_close, NULL, &drv_minfo, NULL

};


static struct qinit drv_winit = {

       drv_put, drv_srv, NULL, NULL, NULL, &drv_minfo, NULL

};


struct streamtab drvinfo = {

       &drv_rinit, &drv_winit, NULL, NULL

};
```

Installation Structures and Configuration Routines

This section describes statically linked modules. For more information on DLKM, refer to the *"DLKM STREAMS" on page 119* . All the kernel modules and drivers on HP-UX are required to define HP-UX specific entry points for installing modules and drivers into the kernel. These entry points are called when a module or driver is installed in the kernel. Entry points register the kernel module_defined in the following structures:

- `streams_info_t`
- `drv_info_t`
- `drv_ops_t`

  These structures can be defined in a `.c` or `.h` file.
- Module metadata

  The module metadata must be defined in a separate file for every STREAMS module and driver. Refer to the *HP-UX Driver Development Guide* for details.

The streams_info_t Structure

The `streams_info_t` structure must be defined for both STREAMS modules and drivers. This structure contains the following fields:

```
char *name;                    /* name of the driver or a module */
int inst_major;                /* major number for driver */
struct streamtab inst_str_tab; /* current streams tab entry */
unsigned int inst_flags;       /* stream flags */
int  inst_sync_level;          /* Synchronization Level */
char inst_sync_info[FMNAMESZ+1]; /* Elsewhere sync param. */
                               /* Set NULL string if not used. */
```

`name`	Points to a string containing the name of the driver. This value must match the name specified in the module metadata.
`inst_major`	Assign the major number to -1 for dynamic assignment. If the major number is manually assigned, you must assign a unique number. Ensure that the `inst_major` matches with the contents of the `c_major` field in `drv_info_t`. For STREAMS modules, set to -1.
`inst_str_tab`	The `streamtab` data structure associated with driver or module.
`inst_flags`	The `inst_flags` can contain the bitwise OR of the following flags:

```
STR_IS_DEVICE      0x00000001    /* For STREAMS driver. */
STR_IS_MODULE      0x00000002    /* For STREAMS module. */
MGR_IS_MP          0x00000008    /* Driver/module is MP safe
                                    Defined in <sys/conf.h>. */
STR_SYSV4_OPEN     0x00000100    /* Supports SVR4 style open()
                                    and close() routines */
STR_MP_OPEN_CLOSE  0x00001000    /* Indicates open() and close()
                                    routines are MP-safe. */
```

`inst_sync_level`

The synchronization level used in the `streams_info_t` structure is defined in `<sys/stream.h>`. The synchronization level determines the level of concurrent access to the module and driver in STREAMS/UX.

The supported synchronization levels are as follows:

`SQLVL_NOSYNC`	Nosync level synchronization. This provides the maximum level of parallelism. Multiple threads can independently access the read and write queues.
`SQLVL_QUEUE`	Queue level synchronization. Two threads can independently access the read and write queues.
`SQLVL_QUEUEPAIR`	Queue-pair level synchronization. Only one thread can access the read and write queues for each instance of a driver or module.
`SQLVL_DEFAULT`	Default level synchronization. It is the same as `SQLVL_QUEUEPAIR`.

SQLVL_MODULE	Module level synchronization. All the instances of modules/drivers are synchronized.
SQLVL_ELSEWHERE	Synchronize with respect to the group of co-operating modules/drivers which access each others data. The group name is specified in the inst_sync_info.
SQLVL_GLOBAL	Provides Global synchronization. No more than one thread of execution can access modules/drivers under this level.

inst_sync_info

Used to provide the name of the driver/module/group in case of SQLVL_ELSEWHERE synchronization. If not used, set NULL string.

The drv_info_t Structure

All STREAMS drivers must define drv_info_t structure and must be allocated statically. This structure is not applicable for STREAMS modules. The structure is defined in <sys/conf.h>.

The drv_info_t has the following fields:

```
char        *name;              /* Name of driver                    */
char        *class;             /* Device class ("disk","tape", etc.) */
ubit32      flags;              /* Device flags (see below)          */
int         b_major;            /* Block device major number         */
int         c_major;            /* Character device major number     */
cdio_t      *cdio;              /* Pointer to cdio(CDIO4) structure   */
void        *gio_private;       /* Additional GIO information        */
void        *cdio_private;      /* Additional CDIO information       */
```

name	Points to a string containing name of the driver. This should match the name specified in module metadata.
class	Pointer to a string containing the name of the class that the driver is in. For interface drivers, instances of a card are counted within each class when they are identified at kernel boot time. Instance numbers are unique within a device class.
flags	Contains the bitmask of flags that describe the module/driver.

DRV_CHAR	Character device driver.
DRV_BLOCK	Block device driver.
DRV_SCAN	Driver supports bus scanning.
DRV_MP_SAFE	MP aware driver and provides its own multi-processing protection.
DRV_SAVE_CONF	Save configuration information to /etc/ioconfig. This file retains potentially volatile information such as dynamic major numbers and card instance numbers across reboots.

b_major	Set this field to -1. STREAMS drivers or modules are not block device drivers.

`c_major`	Assign the major number or set it to -1 for dynamic assignment. If the major number is manually assigned, it is the responsibility of the driver developer to assign a unique number.
`cdio`	Set this field to NULL.
`gio_private`	Set this field to NULL.
`cdio_private`	Set this field to NULL.

The drv_ops_t Structure

This structure is applicable only for STREAMS drivers. This must be allocated statically.

The `drv_ops_t` has the following fields.

```
        d_open_t        d_open;

        d_close_t       d_close;

        d_strategy_t    d_strategy;

        d_dump_t        d_dump;

        d_psize_t       d_psize;

        int             (*reserved0) __(());

        d_read_t        d_read;

        d_write_t       d_write;

        d_ioctl_t       d_ioctl;

        d_select_t      d_select;

        d_option1_t     d_option1;

        pfilter_t       *pfilter;

        int             (*reserved1) __(());

        int             (*reserved2) __(());

        aio_ops_t       *d_aio_ops;

        int             d_flags;
```

The `d_flags` field is the only valid field for STREAMS drivers. STREAMS drivers can use the `d_flags` field. All the other fields in `drv_ops_t` are ignored set to NULL.

The d_flags Field

`C_ALLCLOSES`	If this flag is set, it is the responsibility of driver to keep track of the information about the number of opens and closes for this device.
	If this flag is set, STREAMS sends the `M_CLOSE` message downstream and waits for a reply from the driver. The stream is dismantled if the reply (`M_CLOSE_REPLY`) for the `M_CLOSE` message has `MSGLASTCLOSE` set in the `b_flag`.
`C_CLONESMAJOR`	When this flag is set it indicates that driver supports cloning method where drivers will be able to encode information in their minor numbers. The driver `open()` routine checks for the clone minor number. The driver routine allocates a minor number and returns a new device number to the caller. See the *"Cloning" on page 72* section for more details.

Configuration Routines

The following are the configuration routines for drivers:

`*driver*_install()`:

All STREAMS drivers and modules that are to be statically built into the kernel must implement a `driver_install()` function. The variable is the name of the module or driver as defined in the `module_info` structure. The driver must match the name specified in `drv_info_t/streams_info_t` for drivers.

STREAMS Drivers:	STREAMS drivers must call `install_driver()` and `str_install()` to register the driver with the system and populate the system structures and tables.
STREAMS Modules:	STREAMS modules must call `str_install()` to register the module with the system and populate the STREAMS subsystem tables. The `install_driver()` routine is not applicable for STREAMS modules.

`install_driver()`

Initializes the `cdevsw[]` table with the `d_flag` and `drv_info_t` corresponding to the driver being configured.

`str_install()`

1. Initializes the entry points specified in the `cdevsw[]` table with the STREAMS-specific entry points.
2. Updates `dmodsw` switch tables for a STREAMS driver or the `fmodsw` for the STREAMS modules. These tables are internal to STREAMS subsystem.

Drivers can define additional configuration routines if required, for example, interface drivers (LAN, SLIP, etc need to define *driver*`_attach()`, *driver*`_init()`, etc.). Refer to the *HP-UX Driver Development Guide* for more details on how to write a driver.

The *driver*`_install()` routine must return 0 if the driver is installed successfully, and return the return value of `install_driver()` or `str_install()` in case of failure.

The following code snippet illustrates the installation structures and configuration routines for a dummy STREAMS driver named `drv`:

```
static streams_info_t drv_str_info = {

        "drv",

        -1,

        { &drv_rinit, &drv_winit, NULL, NULL },

        STR_IS_DEVICE | MGR_IS_MP | STR_SYSV4_OPEN,

        SQLVL_QUEUEPAIR,

        ""

};


static drv_info_t drv_drv_info = {

        "drv",

        "pseudo",

        DRV_CHAR | DRV_PSEUDO | DRV_MP_SAFE,
        -1,

        -1,

        NULL,
        NULL,
        NULL


};


drv_opt_t drv_drv_ops ={

        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,


        0,

};


int

drv_install()

{

        int retval;


        if ((retval = (install_driver(&drv_drv_info,

                       &drv_drv_ops))) !=0) {

            return retval;

        }

        if ((retval = str_install(&drv_str_info)) !=0) {

            (void)uninstall_driver(&drv_drv_info);

            return retval;

        }


        return 0;
}
```

NOTE The *driver*_`install()` routine must never sleep.

Entry Points

This section explains the entry points for drivers and modules in detail.

Open

The open routine for a driver is called when the device is first opened by the user process with an open(2) system call. The module open routine is called when the module is pushed onto the stream with I_PUSH IOCTL command or with the autopush utility.

For SVR 4 compliance, the open routine has the following definition:

```
int drv_open(queue_t *q, dev_t *devp, int oflag, int sflag, cred_t *credp);
```

For SVR 3.2 compliance, the open routineis as follows:

```
int drv_open(queue_t *q, dev_t devp, int oflag, int sflag, cred_t *credp);
```

q is the pointer to the driver/module's read queue. devp points to the device pointer in SVR4 open(2). devp is the device number in SVR 3.2 open(2). oflag are the flags used by the application in the open(2) system call, sflag indicates the type of STREAMS open to perform and credp point to the processes credentials. On success the open routine should return 0 and an error number on failure.

Valid sflag values are defined in <sys/stream.h> and are as follows:

0	— Non-cloneable driver open
CLONEOPEN	— Cloneable driver open
MODOPEN	— Module open

Do not call a driver with the sflag set to MODOPEN. But, as a driver can be configured as both module and driver, appropriate sflag should be used in the open routine. i.e., when it is configured as driver sflag should be set to 0 or CLONEOPEN and when it is configured as module sflag should be set to MODOPEN.

In the open routine the module or driver allocates and initializes its resources. For same minor device, the driver open routines are serialized and only one open routine will be allowed to process at a time. Usually, the first open of the device will allocate the structures and subsequent open routines will have very little processing to do if any. Modules and drivers can sleep in the open routine at an uninterruptible priority level or with PCATCH set, as it has user context.

Close

The close routine for a driver is invoked on the last close of the device. The close routine for a module is invoked when it is popped from the stream as a result of a I_POP IOCTL command or closing of the stream.

For SVR 4 compliance, the close routine has the following definition:

```
int drv_close(queue_t *q, int oflag, cred_t *credp);
```

For the SVR 3.2 compliance, the close routine is as follows:

```
int drv_close(queue_t *q, int oflag);
```

Where q is the pointer to the driver or modules read queue, oflag is the bitmask of flags indicating the file status and credp point to the processes credentials. On success the close routine should return 0 and it returns an error value on failure, even though the return value from the close routine is currently ignored by STREAMS.

In the close routine the module or driver should free any resources it allocated in the open path and clear any pending timeouts or `bufcalls`. The close routine must take appropriate action for messages still left in the module or driver queue. For drivers the read-side queue can have messages resulting from device interrupts. Driver developers need to make sure these messages are handled properly prior to actually closing the driver. The `close` routine can sleep at an uninterruptible priority level or with `PCATCH` set, as it has user context.

Ioctl

The `ioctl` (2) system call is used for performing I/O control operations on a character device. For traditional UNIX character I/O devices the IOCTL calls are handled transparently by the kernel, that is, the `ioctl` calls on a device are handled by the driver for that device. STREAMS IOCTLs are an addition to the regular UNIX character input/output mechanism. They differ from the UNIX I/O mechanism in following manner:

- A stream may have multiple modules at any point, and each module may define its own IOCTL commands. Thus, the `ioctls` that can be used on a Stream can very depending on the modules present on the stream.
- The stream head processes a large number of stream head `ioctl` commands that are independent of any module or driver present on the stream. These `ioctl` commands are described in `streamio` (7).
- When modules and drivers receive information associated with an `ioctl`, there is no user context pertaining to the `ioctl` call as the information is received in the form of a message in the put procedure. This prevents modules and drivers to perform `copyin()` or `copyout()` operations and associate any kernel data with currently running process.

The most straightforward examples of STREAMS IOCTLs are for commands like `I_PUSH` and `I_POP` where there is no data sharing between user process and a stream. These IOCTLs are Stream Head IOCTLs and are mainly described in *streamio* (7). But, for other IOCTLs where the user process needs to share data between user-space and kernel-space, the `I_STR` IOCTL and the transparent IOCTL mechanism provide the additional control. The `I_STR` IOCTL mechanism uses only a single pair of messages to share the data between user-space and kernel-space and is described in *"I_STR IOCTL Processing" on page 84* section. The transparent mechanism may use multiple pair of messages to share the data between user-space and kernel-space as described in *"Transparent IOCTL Processing" on page 87* section.

General IOCTL Processing

Most of the IOCTL commands described in *streamio* (7) are processed by the stream head and are not sent downstream for further processing by a module/driver. Commands that require further processing by a module or driver and commands unrecognized by the stream head are sent downstream in the form of an `M_IOCTL` message, created by the stream head. Each of the module which recognizes the IOCTL command in the `M_IOCTL` message must take the required action.

In general, the IOCTL processing requiring an action from a module/driver can be described as follows.

Stream head blocks the user process issuing the `ioctl` (2) system call, creates the `M_IOCTL` message and sends it downstream. The user process remains blocked on the `ioctl` (2) system call until one of the following conditions occur:

- A module or driver on the stream, recognizes the IOCTL command and responds with a positive acknowledgement (`M_IOCACK` message) or a negative acknowledgement (`M_IOCNAK` message).
- The IOCTL request times out.
- The user process interrupts the IOCTL request.
- An error occurs.

The timeout value for STREAMS IOCTL is infinite except for `I_STR` IOCTLs where a user can specify a timeout value.

Modules must pass an unrecognized `M_IOCTL` message to the next component on the stream without any modifications to the message.

Drivers must respond with an `M_IOCNAK` message to the stream head for an unrecognized `M_IOCTL` message.

At any point in time only one `ioctl` (2) system call is active on the stream head.

An `M_IOCTL` message consists of an `M_IOCTL` message block followed by zero or more `M_DATA` blocks. The `M_IOCTL` message block contains an `iocblk` structure defined in `<sys/stream.h>` and has the following fields:

```
        int             ioc_cmd;        /* ioctl command type           */

        cred_t *        ioc_cr;         /* pointer to full credentials  */

        uint            ioc_id;         /* ioctl id                     */

        uint            ioc_flag;       /* see flag values below        */

        ioc_pad         ioc_cnt;        /* count of bytes in data field */

        int             ioc_error;      /* error code                   */

        int             ioc_rval;       /* return value                 */
```

The `ioc_cmd`, `ioc_cr`, `ioc_uid` and `ioc_id` fields of the `iocblk` structure must not be modified by modules or drivers.

The `ioc_error` is an optional field that can be used for setting error codes to accompany an `M_IOCNAK` or `M_IOCACK` message.

The `ioc_rval` can be used to return a value to user application.

If a module or driver detects an error while processing the `M_IOCTL` message, it must send a negative acknowledgement (`M_IOCNAK`) upstream. For an `M_IOCNAK` message no data or return value can be sent to user. If `ioc_error` is set to zero, the stream head will return `EINVAL` to user indicating `ioctl` (2) system call failure. Alternatively, the driver can specify a different `ioc_error` value if desired. can be used to return a value to user application.

For the positive acknowledgment (`M_IOCACK`) message a return value can be sent to the user along with the `M_IOCACK` message. For the `I_STR` IOCTLs the return value can be sent in `ioc_rval` field and for transparent IOCTLs a more general mechanism is used as explained in the *"Transparent IOCTL Processing"* section.

If stream head does not receive an `M_IOCACK` or `M_IOCNAK` message in response to an `M_IOCTL` message, it will block all IOCTL calls except for `I_STR` IOCTLs. For `I_STR` IOCTLs if an `M_IOCACK` or `M_IOCNAK` message is not received as a response to an `M_IOCTL` message, it will fail when the timeout expires and is blocked when the timeout is infinite.

I_STR IOCTL Processing

A user process can make `I_STR` IOCTL call325466665555555555555555555555555555555555555555555555555555555547777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444s by setting the *cmd* parameter to `I_STR`, and the *arg* parameter to a buffer in user space of type `strioctl` as shown:

```
if (ioctl (fd, I_STR, &strioctl) < 0) {

   perror("I_STR ioctl failed");

   exit();
}
```

The structure `strioctl` is defined in `<stropts.h>` and contains the following fields:

```
 int     ic_cmd;         /* downstream command */
        int     ic_timout;      /* ACK/NAK timeout */
        int     ic_len;         /* length of data arg */
        char *  ic_dp;          /* ptr to data arg */
```

`The ic_cmd` field describes the command intended for module or driver, `ic_timout` specifies the number of seconds the `I_STR` request will wait for an acknowledgement before timing out. The `ic_len` field is the length of the data buffer and `ic_dp` field points to the data buffer.

The stream head generates an `M_IOCTL` message for each `I_STR` IOCTL it receives. The attributes of the `strioctl` structure provided by the user are copied into an `iocblk` structure and are attached as the data buffer of the `M_IOCTL` message block. The actual user data is then attached as zero or more `M_DATA` messages to the `M_IOCTL` message block.

The packaged `M_IOCTL` message is then sent downstream by the stream head for further processing. The first component in the stream that recognizes the command in `ic_cmd` processes the message and sends a positive or negative acknowledgment message upstream. If the positive/negative acknowledgment is not received by the stream head within `ic_timout` seconds, then the `I_STR` IOCTL will fail. If data needs to be returned by the target module or driver, `ic_dp` must point to a buffer large enough to hold that data, and `ic_len` will be set on return to indicate the amount of data returned.

The following code sample from a `put` procedure of a driver describes the processing of an `M_IOCTL` message generated from an `I_STR` IOCTL request:

```
drvput(q, mp)
queue_t *q;
mblk_t *mp;
{
    struct iocblk *iocp;
    struct drvdata *mydata;

    /* Get driver private data structure pointer */
    mydata = q->q_ptr;

    switch(mp->b_datap->db_type) /* Check message type */
    {
    ...
    ...
    case M_IOCTL:
         iocp = (struct iocblk *) mp->b_rptr;

         /* Check ioctl command in the M_IOCTL message type */
         switch(iocp->ioc_cmd)
         {
         /* valid ioctl recognized by this driver */
         case SET_OPTIONS:
              /* validate user data:
               * (for example count should be exactly of integer size
               * for this driver user data)
               */
              if (iocp->ioc_count != sizeof(int)) {
                 /* Set error value */
                 iocp->ioc_error = EINVAL;
                 goto log_error;
              }

              if (mp->b_cont == NULL) {
                  goto log_error;
              }

              /* Process actual user data from 2nd mesage block */
              set_mydrv_options(mydata, *(int *)mp->b_cont->b_rptr);

              /* Send acknowledgement upstream */
              mp->b_datap->db_type = M_IOCACK;

              /* no data returned to user */
              iocp->ioc_count = 0;
              qreply(q, mp);
              break;

         log_error:
              /* error, send negative acknowledgement */

         default:
              /* unrecognized ioctls, send negative acknowledgement */
              mp->b_datap->db_type = M_IOCNAK;

              /* No data returned for negative acknowledgement messages */
              iocp->ioc_count = 0;
              qreply(q,mp);
              break;
         }
     }
}
```

Transparent IOCTL Processing

The transparent IOCTL mechanism offers data transfer capability well beyond what is offered by the `I_STR` IOCTL mechanism. The `I_STR` mechanism only enables IOCTL commands that are defined in *streamio* (7). Transparent IOCTL allows user applications and modules to define their own IOCTL commands. The only requirement is that the handling of the `IOCTL` command needs to be mutually understood between the user process that generates the IOCTL and the module that handles the IOCTL. Transparent IOCTL processing becomes necessary, for example, when the user data to be shared with a stream is scattered across multiple buffers, or is in complex structures and cannot be linearized into a single data block as required by `I_STR` processing. Because the user-defined message buffer can be complex, it is to be expected that transparent IOCTL commands may take multiple message pairs for complete processing, as opposed to `I_STR` IOCTLs which takes only one message pair. Above all, the transparent IOCTL mechanism offers a way to incorporate the kernel `copyin()` and `copyout()` functions for STREAMS modules, which are not directly possible due to absence of user context for STREAMS modules and drivers in data processing paths. The kernel `copyin()` and `copyout()` functionality is achieved by having the modules and drivers generate `M_COPYIN` and `M_COPYOUT` messages, and send them upstream to the stream head.

Going back to the interface for the *ioctl* (2) system call, a transparent IOCTL call has following semantics:

```
int ioctl (fd, ioc_cmd, &ioctl_struct);
```

Where `fd` is a STREAMS file descriptor, `ioc_cmd` is user-specified IOCTL command and `ioctl_struct` points to a user buffer in a format mutually agreed upon by the user application and module.

NOTE In this section wherever a module is mentioned it represents a module or driver.

The following steps are generally involved in transparent IOCTL processing:

1. The user application issues an *ioctl* (2) request with a user-defined `IOCTL` command.
2. The stream head receives the request, does not recognize the `IOCTL` command, assumes it is a transparent IOCTL call and accordingly creates an `M_IOCTL` message with a special constant TRANSPARENT set in the `ioc_cnt` field of the `iocblk` structure and `ioc_cmd` field is set to the user-specified `IOCTL` command. The special constant TRANSPARENT is used to recognizes the transparent nature of the `M_IOCTL` message when received by modules. Also, note that the `M_IOCTL` message for transparent processing consists of an `M_IOCTL` message block followed by one `M_DATA` message block containing the four bytes making up the third argument to the *ioctl* (2) system call. This is different from `I_STR` processing where the message may contain zero or more `M_DATA` blocks.
3. The general rules for `M_IOCTL` processing apply to transparent processing. The first module to recognize the `IOCTL` command (set in `ioc_cmd`) will process it. This module also recognizes the transparent nature of the `M_IOCTL` because of the `ioc_cnt` attribute.
4. If the module requires user data, it creates an `M_COPYIN` message and sends it upstream. The `M_COPYIN` message contains the address of user data to copy in and number of bytes requested. The stream head upon receiving the `M_COPYIN` message, issues a `copyin()` request to user-space. The response from the `copyin()` is sent as a `M_IOCDATA` message to the module below, with indication of its success/failure encoded in the `M_IOCDATA` message.
5. The module receives the `M_IOCDATA` message and processes it. It may reuse the `M_IOCDATA` message contents to request another `M_COPYIN` from stream head. The module may send as many `M_COPYIN` messages to the stream head as necessary to get all the required user data.
6. When all the user data has been received and the IOCTL processed, the module sends a positive acknowledgment (`M_IOCACK`) message back to the stream head (or an `M_IOCNAK` message, in case of error).

7. If the module needs to send data back to the user, a M_COPYOUT message is created by the module and sent to the stream head. The stream head executes a copyout() function that copies the data from the M_COPYOUT message into the user buffer. The M_COPYOUT message contains the address of user data, size and the data. The M_IOCDATA response for the M_COPYOUT will not contain data but only an indication of success or failure.

The format of a M_COPYIN message is one message block of type M_COPYIN, whose data buffer contains a copyreq structure as defined. The format of a M_COPYOUT message is one message block of M_COPYOUT, linked to one more M_DATA blocks containing data to be copied to the user's buffer. The data buffer in a M_COPYOUT message also contains a copyreq structure. The format of M_IOCDATA message is one message block of type M_IOCDATA, linked to zero or more M_DATA blocks. The data buffer of M_IOCDATA contains a copyresp structure as defined.

The copyreq structure in M_COPYIN/M_COPYOUT messages used to communicate requests from the modules to stream head is defined in <sys/stream.h> and contains the following fields:

```
int            cq_cmd;        /* command type == ioc_cmd */
cred_t *       cq_cr;         /* pointer to full credentials */
uint           cq_id;         /* ioctl id == ioc_id */
int            cq_flag;       /* reserved */
mblk_t *       cq_private;    /* module's private state info */
ioc_pad        cq_ad;         /* address to copy data to/from */
uint           cq_size;       /* number of bytes to copy */
```

If a new copyreq message is allocated for M_COPYIN/M_COPYOUT request, modules must copy the values of cq_cmd, cq_cr and cq_id from the M_IOCTL or M_IOCDATA message into respective fields of the new message. These fields should not be modified. The cq_ad refers to the address of the data buffer from which data needs to be copied in for an M_COPYIN message, while it refers to the address of the buffer from which data needs to be copied out for a M_COPYOUT request. The cq_size specifies the number of bytes to be copied. Both cq_ad and cq_size values need to be set by the modules. The cq_private can be set by the modules to get their state information.

The copyresp structure in M_IOCDATA used to communicate the results of copyin()/copyout() from the stream head back to the modules is defined in <sys/stream.h> and contains the following fields:

```
int            cp_cmd;        /* command type == ioc_cmd */
cred_t  *      cp_cr;         /* pointer to full credentials */
uint           cp_id;         /* ioctl id == ioc_id */
uint           cp_flag;       /* flag values */
mblk_t *       cp_private;    /* module's private state info */
ioc_pad        cp_rv;         /* 0 = success */
```

The values of cp_cmd, cp_cr, cp_id, cp_flag and cp_private refer to the cq_cmd, cq_cr, cq_id, cq_flag and cq_private fields respectively in the copyreq structure of M_COPYIN/M_COPYOUT messages. In response to the M_COPYIN message, the M_DATA portion of the M_IOCDATA contains the data copied in from the user buffer. In response to a M_COPYOUT message, there are no M_DATA blocks present, instead an indication of whether or not copy succeeded is returned through cp_rv. The cp_rv is set to zero in case of success and it is set to error number in case of failure.

A module may intersperse M_COPYIN and M_COPYOUT messages as required. The only requirement is that only one such request may be pending at any time, i.e., prior to issuing the next M_COPYIN or M_COPYOUT, the module must wait until it receives the M_IOCDATA message from a previous copy request.

The stream head converts the `M_COPYIN` and `M_COPYOUT` messages into `M_IOCDATA` messages by changing the type of the message, so it is absolutely necessary that the `copyreq` structure be properly formed and initialized. Also, the `iocblk`, `copyreq` and `copyresp` structures all overlay one another.

`M_COPYIN`, `M_COPYOUT` and `M_IOCDATA` are high-priority messages and are used for communication between the module that created them and the stream head. So, no other modules between these two should process these messages.

Transparent IOCTL Example

The user issues the transparent IOCTL as follows:

```
ioctl(fd, I_TRANSPARENT, &bufadd);
```

The module recognizes the transparent IOCTL and processes the IOCTL as follows:

```
struct transparent_data { int cmd; int buflen; caddr_t bufaddr; };
struct state { int st_state; struct transparent_data st_data; };

modwput(q, bp)
queue_t *q;
mblk_t *bp;
{
   struct iocblk *iocbp;
   struct copyresp *csp;
   struct copyreq  *cq;
   struct state *stp, *tmp;

   switch(bp->b_datap->db_type) {
   ...
   ...
   case M_IOCTL:
        iocbp = (struct iocblk *)bp->b_rptr;
        switch(iocbp->ioc_cmd) {
        ...
        ...
        case I_TRANSPARENT:
             if (iocbp->ioc_count == TRANSPARENT) {
                /* Reuse M_IOCTL block for M_COPYIN */
                cq = (struct copyreq *)bp->b_rptr;

                /* Get user structure address from linked M_DATA block */
                cq->cq_addr = (caddr_t) *(long *)bp->b_cont->b_rptr;

                /* free linked message */
                freemsg(bp->b_cont);
                bp->b_cont = nilp(mblk_t);
                /* Allocate state buffer */
                if ((tmp = allocb(sizeof(struct state), BPRI_LO)) == NULL) {
                    bp->b_datap->db_type = M_IOCNAK;
                    iocbp->ioc_error = EAGAIN;
                    qreply(q, bp);
                    break;
                }
                tmp->b_wptr += sizeof(struct state);
                stp = (struct state *)tmp->b_rptr;
                stp->st_state = GETSTRUCT;
                cq->cq_private = tmp;
                cq->cq_size = sizeof(struct transparent_data);
                cq->cq_flag = 0;
                bp->b_datap->db_type = M_COPYIN;
                bp->b_wptr = bp->b_rptr + sizeof(struct copyreq);
             } else {
                /* Send negative acknowledgement */
                bp->b_datap->db_type = M_IOCNACK;
                iocbp->ioc_error = EINVAL;
             }
             qreply(q, bp);
             break;
        default:
             /* Unknown message, so pass to next component */
             putnext(q, bp);
             break;
        }

   case M_IOCDATA:
        iocbp = (struct iocblk *)bp->b_rptr;
        csp = (struct copyresp *)bp->b_rptr;
        if (csp->cp_cmd == I_TRANSPARENT) {
           if (csp->cp_rval) { /* GETSTRUCT failure */
              if (csp->cp_private) /* state structure */
                 freemsg(csp->cp_private);
              freemsg(bp);
              break;
           }
           stp = (struct state *)csp->cp_private->b_rptr;
           switch( stp->st_state ) {
           case GETSTRUCT:
                stp->st_data = *(struct transparent_data *)bp->b_cont->b_rptr;
                freemsg(bp->b_cont);
```

```
          bp->b_cont = nilp(mblk_t);
          /* reuse M_IOCDATA to copyin data */
          bp->b_datap->db_type = M_COPYIN;
          cq = (struct copyreq *)bp->b_rptr;
          cq->cq_size = stp->st_data.buflen;
          cq->cq_addr = stp->st_data.bufaddr;
          cq->cq_flag = 0;
          stp->st_state = GETINDATA; /* next state */
          qreply(q, bp);
          break;

     case GETINDATA: /* data successfully copied in */
          /* process input */
          ...
          ...
          /* return output */
          bp->b_datap->db_type = M_COPYOUT;
          cq = (struct copyreq *)bp->b_rptr;
          cq->cq_size = stp->st_data.buflen;
          cq->cq_addr = stp->st_data.bufaddr;
          cq->cq_flag = 0;
          stp->st_state = PUTOUTDATA; /* next state */
          qreply(q, bp);
          break;

     case PUTOUTDATA:
          freemsg(csp->cp_private); /* state structure */
          csp->cp_private = nilp(mblk_t);
          bp->b_datap->db_type = M_IOCACK;
          iocbp->ioc_error = 0; /* may have been overwritten */
          iocbp->ioc_count = 0; /* may have been overwritten */
          iocbp->ioc_rval = 0;  /* may have been overwritten */
          qreply(q, bp);
          break;

     default:
          /* Unknown state: This can't happen but in case */
          freemsg(bp->b_cont);
          bp->b_cont = nilp(mblk_t);
          bp->b_datap->db_type = M_IOCNAK;
          iocbp->ioc_error = EINVAL;
          qreply(q, bp);
          break;
          }
        } else {
           /* M_IOCDATA not for us , so pass to next component */
           putnext(q, bp);
        }
        break;
   default:
        /* Unknown message, so pass to next component */
        putnext(q, bp);
        break;
   }
}
```

Put Procedure

Modules and drivers need to use the put or service procedure entry points for providing all the message processing logic. The put procedure is used for immediate processing of the message and is required for all queues in a stream, with a possible exception of driver read-queue.

The put procedure takes as input a queue pointer and a message pointer and has following interface:

```
int drv_put (queue_t *q, mblk_t *mp);
```

The return value from the put procedure is ignored.

A put procedure processes one message at a time. It cannot call sleep or any routines that block since it may not have user context. The put procedure at a minimum should handle high-priority messages and flush messages to avoid delayed processing of these messages. For all other messages it can either do deferred processing by enqueuing the message using `putq()` utility or immediately process it.

When a message is passed to a driver it must do one of the following three things in its put procedure as an action on the message; free the message, send the message back upstream or enqueue the message on its queue for deferred processing by the driver service procedure.

When a module receives the message, it should pass the message to the next component on the stream or send it back upstream as one of the action on the message for immediate processing of the message. It can also enqueue the message if it wants to defer the processing of the message.

A driver must always free an unrecognized message with an exception of `M_IOCTL` messages which must be replied with an `M_IOCNAK` message. A module should pass an unrecognized message to the next component in the stream.

Service Procedure

The service procedure entry point is mainly used for deferred processing of the message apart from its use for recovering from buffer allocation failures and implementing flow control. The use of service procedure for a processing message allows minimum processing of the message on **Interrupt Context Stack** (ICS) and avoids stack overflow caused due to multiple put procedure calls on the STREAMS stack.

The service procedure is optional for a queue but is required for modules and drivers that place their message on their queue and/or implement flow control.

The service procedure takes as input a queue pointer and has following interface:

```
int drv_srv (queue_t *q);
```

The return value from the service procedure is ignored.

A service procedure is scheduled to run by placing a message on its queue. When a service procedure is executed by the scheduler, it should process all the messages on its queue. If it cannot process all the messages, then it must ensure that it gets rescheduled so that the remaining messages are processed. One of the rescheduling mechanisms provided by streams is flow-control, where the service procedure is back-enabled if the queue to which it wants to send the message is flow-controlled. In this scenario, the service procedure cannot empty its queue due to flow-control condition and needs to put the message back on its queue and need not worry about rescheduling as the flow-control back enabling mechanism would enable the service procedure when the flow-control conditions are removed.

For various other reasons where the service procedure cannot process all the messages from its queue, `bufcall` and timeouts can be used for rescheduling the service procedure.

The rules that apply to put procedure for sleeping and unknown message types (including IOCTL commands) also apply to service procedures. In addition service procedures must never place high-priority messages back on their queues. This will result in infinite loop because when a high-priority message is placed on a queue its service procedure is scheduled.

Interrupt Service Routine

STREAMS drivers associated with real hardware device, should specify a interrupt service routine entry point for handling interrupts like regular device drivers. Drivers should specify their interrupt handler routine in the `wsio_intr_alloc()` routine which is executed in *drv*`_attach()` or *drv*`_if_init()` routine. See the *HP-UX Driver Development Guide* for more details on driver interrupt handling.

The interrupt service procedure has the following interface:

```
int drv_isr (long arg1);
```

Drivers are usually replaced with the driver's name and `arg1` is the argument passed in the `wsio_intr_alloc()` routine along with driver's interrupt service procedure name.

Since, the interrupt service procedure is executed in interrupt context and does not have any user context associated with it, it should not call `sleep()` or any routines that would block. Interrupts service routines are called with processor priority level elevated and they block interrupts at same or lower level interrupts. The processing in these routines should be minimal to decrease the performance degradation that can occur due to blocking of other interrupts.

Flush Handling

The Flush operation involves removing of messages from read queue or write queue or both the queues. It can be initiated by a user process or by a module/driver.

The `M_FLUSH` message is used in flush operation. All drivers and modules must handle the `M_FLUSH` messages. STREAMS provides two utilities — `flushq()` and `flushqband()`, for drivers and modules to flush an entire queue or just a specified band.

Stream head generates `M_FLUSH` message and send downstream upon receiving a `I_FLUSH` or `I_FLUSHBAND` IOCTL. Modules/Drivers can initiate the flush operation by sending `M_FLUSH` upstream or downstream as appropriate by setting the flags to indicate if read queue or write queue or both the queues to be flushed:

The first byte in the data buffer of an `M_FLUSH` message contains flags that determines the type of requested flush operation. It can have the following values. If `FLUSHBAND` is set, then the second byte contains the priority band value and only messages in this priority band are flushed.

`FLUSHR`	Flush the read queue.
`FLUSHW`	Flush the write queue.
`FLUSHRW`	Flush both read and write queues.
`FLUSHBAND`	Flush a specified priority band.

M_FLUSH Message Processing

Stream Head	A stream head receiving an `M_FLUSH` message will check the flags first. If `FLUSHR` is set, it flushes its read queue and turns off the `FLUSHR` bit. Following that, if `FLUSHW` is set, stream head will turn the message around and send it downstream. If `FLUSHW` is not set, it frees the message instead.
Module	A module must flush read or write or both the queues based on type of the flags specified in the first byte of the data buffer of an `M_FLUSH` message. After processing, it must send it to the next component in the direction of data flow.
Driver	A driver flushes its write queue if `FLUSHW` is set and unsets the `FLUSHW` flag. If `FLUSHR` is not set, driver frees the message. Otherwise, it flushes its read queue and send the `M_FLUSH` message upstream.

If a driver wants to flush the entire stream, it would flush its read/write queues. Following that it generates an `M_FLUSH` message with `FLUSHRW` set in the first byte of the data buffer. This `M_FLUSH` message is sent upstream for modules/stream head processing.

If a module wants to flush the entire stream, it would generate `M_FLUSH` message with `FLUSHW` set and sends it upstream. Then, the module generate another `M_FLUSH` message with `FLUSHR` set and send it down the write side of the stream.

Flush Handling in a Pipe

Message flushing in a pipe is complicated in nature because the write queue of one end of stream head is connected to the read queue of the other, and vice versa. As a result, `FLUSHW` and/or `FLUSHR` bits have to switched appropriately to ensure that wrong queues are not flushed. The point of switching is called the midpoint of a pipe. Midpoint in a STREAMS based pipe is where the write queues point to the read queues. For example, when a stream head receives an `M_FLUSH` message with `FLUSHW` set, it turns around on the write side of the stream. Write queues of all the modules will be flushed until it reaches a midpoint where `FLUSHW` is switched to `FLUSHR`. Then onwards, the read queues of all the modules including read queue of stream head will be flushed.

STREAMS framework sets the `MSGNOLOOP` flag in `b_flag` the very first time when an `M_FLUSH` reaches a stream head. This is done to ensure that it is not reflected back to the other side of the stream when this message reaches the other stream head.

The midpoint of a stream can be difficult to determine if modules are being pushed from both ends of the pipe. To make the midpoint deterministic, STREAMS provides the `pipemod` module. This module must be pushed onto a pipe for message flushing to work properly. It defines the midpoint of a stream and can be pushed from either end of a pipe. The only requirement is that `pipemod` must be the first module pushed onto a pipe.

The pipemod module handles only `M_FLUSH` messages. All other messages are passed to the next module in the stream using `putnext()` utility.

Summary of pipemod functionality:

1. If `pipemod` receives an `M_FLUSH` message with `FLUSHW` bit set, it shuts off the `FLUSHW` and sets `FLUSHR` and pass it to the next module.
2. If `FLUSHR` bit is set in an `M_FLUSH` message, then pipemod unsets the `FLUSHR` bit and sets the `FLUSHW` bit before passing to the next module.
3. If `FLUSHRW` is set then this message is not processed, instead it is passed to the next module.

Design Guidelines

Some of the common design guidelines for STREAMS modules and drivers have been explained here:

- STREAMS modules and drivers are not associated with any process and do not have a process or a user context (except during open and close. Therefore modules and drivers cannot access the information from the `u_area` of a process.
- Every STREAMS module and driver must process the `M_FLUSH` message according to the value of the argument passed in the message.
- The contents of the data block (`dblk_t`) of a duped message, i.e., a message with the reference `count` (`db_ref`) greater than one must not be changed by STREAMS drivers and modules.
- The modules and drivers should manipulate the queues and manage message buffers using the STREAMS utilities provided in *Appendix B, "STREAMS Utilities Supported by HP-UX."*
- The modules and drivers should refrain from imposing alignment or formatting rules on data in a `M_DATA` message.
- A module or driver's synchronization level determines the entities with which it can share data. The synchronization level determines which queues a module or driver can pass to STREAMS utilities and also the entities with which the modules and drivers can share the STREAMS queues.

  For example, if a module uses queue pair synchronization, the write-side put procedure can call `insq()` to insert a message onto the module's read queue. But, if the module uses queue synchronization, the write-side put procedure can only call `insq()` to insert messages onto the write queue.

  In general, a put or service procedure can only pass its own queue or queues belonging to entities with which it can share data. The restricted utilities are `backq`, `bcanputnext`, `canputnext`, `flushband`, `flushq`, `freezestr`, `getq`, `insq`, `putbq`, `putnext`, `putnextctl`, `putnextctl1`, `putnextctl2`, `putq`, `qreply`, `qsize`, `rmvq`, `SAMESTR`, `strqget`, `strqset`, and `unfreezestr`. The `putq` utility is not restricted when it is passed a driver's read queue or a lower mux's write queue. Any put or service procedure can call `putq` if it passes a driver's read queue or a lower mux's write queue. However, `putq`'s caller must guarantee that the queue passed in is still allocated.

  Some STREAMS utilities, such as `canput`, are commonly passed a parameter of the form `q->q_next`. These routines are restricted in a different way from those previously listed. A put or service procedure can only pass its own queue's `q_next` field or the `q_next` field of queues belonging to entities with which it can share data. These requirements apply to `bcanput`, `canput`, `put`, `putctl`, `putctl1`, `putctl2`, and `streams_put`. These utilities are not restricted when they are passed a parameter of the form `q`, except that the queue must still be allocated.
- Some restrictions exist for timeout and `bufcall` callback routines as well as non-streams code in the kernel. This software cannot share data structures with STREAMS modules and drivers, unless spinlocks are used to protect critical sections. Also, the code cannot call the following utilities: `backq`, `bcanputnext`, `canputnext`, `flushband`, `flushq`, `freezestr`, `getq`, `insq`, `putbq`, `putnext`, `putnextctl`, `putnextctl1`, `putnextctl2`, `qreply`, `qsize`, `rmvq`, `SAMESTR`, `strqget`, `strqset`, and `unfreezestr`.

  Callback routines and non-streams code cannot call `bcanput`, `canput`, `put`, `putctl`, `putctl1`, `putctl2` or `streams_put` if they pass the utility a parameter of the form `q->q_next`. They can call these utilities if they pass a parameter of the form `q` (`q` must be a valid, allocated queue). Callback and non-streams code can call `putq` or the `streams_put` utility only if they pass it a driver's read queue or a lower mux's write queue.

- STREAMS modules and drivers must not call the put and service procedures directly. They must be executed by calling STREAMS utilities such as `putnext`, `put`, `putq`, or `qenable`. They cannot be called using the function pointer stored in the `q_qinfo` structure.
- STREAMS modules and drivers can allocate their own spinlocks to protect data structures. If they do, they should use the lock orders reserved for them in `/usr/include/sys/semglobal.h`: `STREAMS_USR1_LOCK_ORDER`, `STREAMS_USR2_LOCK_ORDER` and `STREAMS_USR3_LOCK_ORDER`.

  Modules and drivers cannot hold spinlocks when calling some STREAMS utilities (see *Appendix B, "STREAMS Utilities Supported by HP-UX,"* for details). To reduce contention and improve performance, the amount of time that the modules and drivers hold the spinlocks should be minimized.
- STREAMS modules and drivers written for HP-UX need to be multiprocessor safe (MP-safe) and specify the flag `MGR_IS_MP` explicitly in their respective `streams_info_t` structures.
- On HP-UX `modmeta` files as mentioned in the *"Installation Structures and Configuration Routines" on page 76* need to be defined for both STREAMS modules and drivers in order to configure them into the kernel.

Rules

A few rules have been established as guidelines for the following routines and procedures.

Open/Close Routines

1. The open and close routines may sleep with a priority `<= PZERO` or with `PCATCH` set in the sleep priority, so as to return to the calling routines when the sleep is interrupted.
2. The open and close routines have a user context and can access the `u_area` (defined in `<sys/user.h>`).
3. The open and close routines should only access the following fields in the `u_area` namely, `u_procp`, `u_ttyp`, `u_uid`, `u_gid`, `u_ruid`, and `u_rgid`.
4. The open and close routines also have access to some attributes `p_pid`, and `p_pgrp` in the process table (defined in `<sys/proc.h>`).
5. The open routine should return zero on success and an error number on failure. When a cloneable STREAMS driver executes its open routine (`sflag` has the `CLONEOPEN` flag set), the device number must be set to an unused device number for that device.
6. If a module or driver wants to allocate a controlling terminal, a `M_SETOPTS` message should be sent to the stream head (by the open routine) with the `SO_ISTTY` flag set.

Handling IOCTLs

1. If a module does not recognize the `M_IOCTL` message, it should forward the message to next component in the stream without any change to the message. If a driver does not understand an IOCTL, it must send a `M_IOCNAK` message upstream.
2. The modules and drivers must not change the `ioc_id`, `ioc_cmd`, `ioc_uid` or `ioc_gid` fields in a `M_IOCTL` message.
3. In addition, the drivers and modules must not change `cp_id`, `cp_cmd`, `cp_uid`, and `cp_gid` fields in `M_IOCDATA` messages and `cq_id`, `cq_cmd`, `cq_uid` and `cq_gid` in `M_COPYIN`/`M_COPYOUT` messages.
4. Modules and drivers should always validate `ioc_count` to see whether the IOCTL is the transparent or the `I_STR` form.

Put Procedure

1. A put procedure must be defined in the `qinit` structure for every queue in a stream to pass messages between the STREAMS components.
2. A put procedure does not have any user context, so it must not sleep and must not call any functions that call `sleep()`.
3. The put procedure cannot access the information in the `u_area` of a process as no user context is associated with it.
4. A put procedure must use the streams utility `putq()` to enqueue a message on its own queue to maintain the consistency of the queue structure. The put procedure of a queue must only call `putq()`, if the queue has a service procedure associated with it.
5. The `putq()` does not process `M_FLUSH` messages. Therefore if `putq()` is specified as the put procedure for a queue in its `qinit` structure, then the service procedure defined for the same queue must process the `M_FLUSH` messages.
6. When a modules/driver's put procedure needs to pass the message to its next component the streams utility `putnext()` must be used. The `putq()` must not be used to place the message on the next components queue directly.
7. Processing data messages by both put and service procedures could lead to messages going out of sequence. The put procedure should check if any data messages were queued before processing the current message.
8. Return codes can be sent with `M_IOCACK`, `M_IOCNAK` and `M_ERROR` messages by a put procedure.
9. Processing too many function calls with the put procedure could lead to an interrupt stack overflow. To avoid such a case, switch to service procedure processing whenever appropriate to switch to a different stack.
10. Appropriate synchronization should be provided for data structures in a module/driver when both the put and service procedures use them.
11. It is strongly recommended that put procedures not place the high priority messages on the queue.

Service Procedure

1. A service procedure does not have any user context, so it must not `sleep()` and must not call any functions that call `sleep()`.
2. The service procedure cannot access the information in the `u_area` of a process as no user context is associated with it.
3. If flow control is desired, a service procedure must be defined. The `canput()` utilities should be used by service procedures before doing a `putnext()` to honor flow control.
4. The service procedure must use the streams utility `getq()` to remove a message from its message queue to maintain the flow control mechanism.
5. The service procedure should process all the messages on its queue. The only exception being if the stream ahead is flow controlled (i.e., `canput()` fails) or if some error condition (for example, memory allocation failure) is encountered. Adherence to this rule is the only guarantee that STREAMS will schedule the service procedure to execute when necessary and to ensure that the flow control mechanism does not fail.

   If a service procedure is written to exit for other reasons, than the driver/module developer must take explicit steps to ensure that the service procedure is reenabled.

6. The service procedure should not put a high priority message back on the queue to avoid getting into an infinite loop.

STREAMS Module

A STREAMS module is essentially a pair of queues and a defined set of kernel-level routines and data structures used to process messages as they flow through them in a stream. A stream may have zero or more modules and **pushing** and **popping** of these modules happens in a **Last In First Out** (LIFO) manner.

Flow Control in Modules

Module flow control is advised and when used helps in limiting the amount of data that can be placed on a modules' queue. STREAMS modules must define a service procedure to utilize the STREAMS flow control mechanism; invoking `canput()`/`canputnext()` before calling `putnext()` and using appropriate high and low water marks for the queues. In addition to `canput()`, the streams utilities `getq()`, `putq()` and `putbq()` are also used in implementing the module's flow control.

In a module implementing the flow control:

- The put procedure queues the data using `putq()` (but forwards all the high priority messages regardless of flow control).
- The `putq()` in turn increments the `q_count` appropriately (sets the `QFULL` flag if `q_count` exceeds the high water mark) and enables the service procedure.
- When the STREAMS scheduler runs the service procedure, data is retrieved by using `getq()`.
- The `getq()` decrements the `q_count` by an appropriate value (unsets the `QFULL` flag if `q_count` drops below the low water mark and enables the nearest back queue with a service procedure).
- The service procedure must verify if the next component in the stream is flow controlled by doing a `canputnext()` and do a `putnext()` if not flow controlled. If `canput()`/`canputnext()` fails, the module should put back the message on it's own queue by doing a `putbq()`.
- If the module ahead is not flow controlled, the service procedure must process all the messages on it's queue before it returns.

Sample Module

The various data structures and routines required to define a STREAMS module have already been described in the previous sections. In this section code examples have been given to substantiate the same.

NOTE There is no major number associated with a module, hence a value of "-1" needs to be specified in a module's `streams_info_t` struct.

Module Declaration

```
/* Sample Module inclusions */


#include <sys/types.h>

#include <sys/errno.h>

#include <sys/stream.h>

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

```
/* Streams data structures for Modules */

int mod_open  __((queue_t *, dev_t *, int, int, cred_t *));
int mod_close __((queue_t *, int, cred_t *));
int mod_rput  __((queue_t * q, mblk_t *));
int mod_wput  __((queue_t * q, mblk_t *));
int mod_rsrv  __((queue_t * q));
int mod_wsrv  __((queue_t * q));

#define MOD_ID  0

static struct module_info minfo =  {
MOD_ID, "MOD", 0, INFPSZ, 65536, 1024
};

static struct qinit mod_rinit = {
mod_rput, mod_rsrv, mod_open, mod_close, NULL, &minfo
};

static struct qinit mod_winit = {
mod_wput, mod_wsrv, NULL, NULL, NULL, &minfo
};

static streams_info_t mod_str_info = {
      "MOD",                                    /* Module name */
      -1,                                       /* major no */
      { &mod_rinit, &mod_winit, NULL, NULL },   /* streamtab */
      STR_IS_MODULE | MGR_IS_MP | STR_SYSV4_OPEN, /* streams flags */
      SQLVL_QUEUEPAIR,                          /* sync level */
      "",                                       /* elsewhere sync name */
};

struct streamtab modinfo = {
       &mod_rinit,
       &mod_winit
};
```

NOTE Setting module ID to “0” lets STREAMS select a unique module ID dynamically when the module gets installed on a system. Make sure that the module name is unique.

Installation Routine

STREAMS modules do not have any `cdevsw`-related information. They only have STREAMS-specific information and this is configured by calling `str_install()` with a defined `streams_info_t`.

```
int
mod_install()
{
    int retval;

    if ((retval = str_install(&mod_str_info)) !=0)
        return retval;  /* failure */

    return 0; /* Success */
}
```

NOTE Any module specific global data structures, spinlocks etc. can be setup and initialized in the installation routine.

Open/Close Routines

As described in the earlier sections the open and close routines associated with a module get invoked when the module is **pushed** onto a stream or when it is **popped** out of a stream.

Based on the module's requirement, a module can:

- Verify that `MODOPEN` is specified in the `sflag`.
- Validate the open flags specified in the `oflag`.
- Validate the user credentials passed through `credp`.

```
int
mod_open(q, dev, oflag, sflag, credp)
queue_t *q;
dev_t   *dev;
int     oflag;
int     sflag;
cred_t  *credp;
{

    mod_priv_t *modp;  /* Pointer to the module's private data */

    /*
     * Allocate the module specific private data to be assigned to
     * the q_ptr
     */
    modp = (mod_priv_t *)kmem_alloc(sizeof(mod_priv_t), M_NOWAIT);
    if (!modp) {
        return 1 ; /* Failure */
    }

    /*
     * Assign the private data structure to both the read and the write
     * side q_ptr's.
     */
    q->q_ptr = WR(q)->q_ptr = modp;

    return 0 ; /* Success */
}
```

Based on a module’s requirement, a module should:

- Validate the flag value specified in the `cflag`.
- Validate the user credentials passed through `credp`.
- Cancel any pending timeout or `bufcall` routines that access data that are deinitialized or deallocated during close.
- Deallocate any resources allocated on open.

```
int
mod_close(q, cflag, credp)
queue_t *q;
int     cflag;
cred_t  *credp
{

     flushq(WQ(q), FLUSHALL);
     /*
      * Free any module specific resources allocated during open for
      * this instance of the module.
      */
     kmem_free((caddr_t)q->q_ptr, sizeof(mod_priv_t));

     /*
      * Assign NULL to both the read and the write side q_ptr's
      */
     q->q_ptr = WR(q)->q_ptr = NULL;

     return 0 ; /* Success */
}
```

Put Procedure

An example of a module's read-side and write-side put procedure are shown:

```
int
mod_rput(q, mp)
queue_t *q;
mblk_t  *mp;
{
     mod_priv_t *modp;

     /* get the module specific data */
     modp = (mod_priv_t *)q->q_ptr;

     if ((mp->b_datap->db_type >= QPCTL) &&
   mp->b_datap->db_type != M_FLUSH )) {
         /*
          * Process the high priority messages immediately
          * and pass it upstream
          */
         ...
         putnext(q, mp);
         return;
     }

     switch (mp->b_datap->db_type) {
         case M_DATA:
              putq(q, mp);
              return;
         case M_PROTO:
              /* process the protocol message */
              ...
              ...
         case M_FLUSH:
      /* process the M_FLUSH message */
      ...
      ...
         default:
              /* Pass the message upstream if the module does not
               * understand it.
               */
putq(q, mp);
```

```
               return;
          }
     }

     int
     mod_wput(q, mp)
     queue_t *q;
     mblk_t  *mp;
     {

          mod_priv_t *modp;

          modp = (mod_priv_t *)q->q_ptr;
          if ((mp->b_datap->db_type >= QPCTL) &&
       ( mp->b_datap->db_type != M_FLUSH )) {
              putnext(q, mp);
              return;
          }

          switch (mp->b_datap->db_type) {
              case M_DATA:
                   putq(q, mp);
                   return;
              case M_PROTO:
                   /* process the protocol message */
                   ...
                   ...
              case M_FLUSH:
           /* process the M_FLUSH message */
           ...
           ...
              case M_IOCTL:
                   struct iocblk * iocp = (struct iocblk *)mp->b_rptr;

                   switch (iocp->ioc_cmd) {
                       /* ioctl commands recognized by the module */
                       case MOD_IOCTL1:
                            /*
                             * process and reply back with a positive
                             * or negative acknowledgment
```

```
                         */
                        ...
                        if (success) {
                            ...
                            mp->b_datap->db_type = M_IOCACK;
                        } else {
                            ...
                            mp->b_datap->db_type = M_IOCNAK;
                        }
                        qreply(q, mp);
                        return;
                   case MOD_IOCTL2:
                        ...
                        ...
                   /* ioctl commands not recognized by the module */
                   default:
                        /* Pass the ioctl message downstream */
                        putnext(q, mp);
                        return;
               }
          default:
               /* Pass the message downstream if the module does not
                * understand it.
                */
  putq(q, mp);
               return;
     }
}
```

Service Procedure

The following example shows a generic service procedure:

```
int
mod_rsrv(q)
queue_t *q;
{

    mblk_t *mp;

    while ((mp = getq(q)) != NULL) {
        /* check for flow control */
        if (!(mp->b_datap->db_type >= QPCTL) &&
              !canputnext(q) {
            putbq(q,mp);
            return;
        }

        /* Process the message */
        switch (mp->b_datap->db_type) {
            case M_DATA:
                 ...
                 ...
        }
     } /* while */
}
```

Module Specific Design Guidelines

In addition to the guidelines listed in the *"Design Guidelines"* section in this chapter, the module developers should follow these guidelines:

- If a module does not understand a message type it must forward the message to the next component on the stream.
- For reusability, module design should stay independent of the underlying device or driver functionality.
- If the module acts on a `M_IOCTL` message, it must send a `M_IOCACK` or `M_IOCNAK` in response to the IOCTL. Instead, if the module does not understand the IOCTL, it must pass the `M_IOCTL` message to the next component in the stream.
- Filter modules pushed between a service user and a service provider must not alter the contents of the `M_PROTO` or `M_PCPROTO` block in messages. The contents of the data block can be modified, however the message boundaries must be preserved.

STREAMS Driver

A STREAMS device driver as mentioned earlier is similar to a regular UNIX character device driver. A STREAMS driver is a necessary part of the stream constituting the stream end. A stream is created when a STREAMS driver is opened. A STREAMS driver can also have multiple streams connected to it. Multiple connections could be a result when more than one minor device of the same driver are opened and in case of multiplexors.

Overview of Drivers

A driver is the software that provides an interface between a device and the operating system. User level programs interact with these devices through system calls. Drivers manage the data going in and out of the the devices and also manage the interrupts generated by the devices.

Based on the type of device the driver controls they can be classified as hardware or software drivers. A hardware driver controls a physical device. A software driver, also known as the pseudo-device driver controls the software, which in turn may be associated with a hardware device or a pseudo device — that has no associated physical device.

In UNIX, the devices are represented as files. Implying that the drivers can support the character-type or block-type access methods.

Finally, depending on the interface used between the drivers and devices, the drivers could be STREAMS-based or non-STREAMS-based.

NOTE STREAMS drivers are always accessed through character special files, so are character based drivers.

Writing Drivers

A driver is different from the regular C applications, in that it executes in the kernel and does not have a `main()` function associated with it. The following general rules apply to all driver development:

- A driver may need a start routine for initialization. This applies more to drivers that control real hardware devices.
- A driver must define `open()` and `close()` routines.
- A driver must have an interrupt handling routine if it controls a hardware device.
- A driver cannot use archive or shared libraries or floating point arithmetic.

All drivers must define specific entry points for initialization, switch table entry and interrupt handlers.

Major and Minor Numbers

In HP-UX device numbers (data type `dev_t`) formed by concatenating the major and the minor numbers is used to identify the particular driver and device that is being accessed. The major number for a device maps to a driver controlling the device and the minor number uniquely identifies the device.

A special device file is created for every device installed on the system, allowing user processes to view the individual devices as files. Major and minor numbers are associated with the special device files and are used by the operating system to determine the actual driver and device to be accessed by the user-level request for the special device file.

A device is then accessed by opening, reading, writing or closing the special device file associated with the device with proper major and minor numbers. The following example lists such special device files:

```
$ ls -l /dev/udp*

crw-rw-rw-   1 root       root        72 0x000033 Dec 18 05:40 /dev/udp

crw-rw-rw-   1 root       root        72 0x000036 Dec 18 05:40 /dev/udp6
```

Where, `udp` and `udp6` are two different devices with major number as `72` and unique minor numbers `0x000033` and `0x000036` respectively.

Major numbers for the hardware devices are typically assigned by the system at boot time or by utilities like `install_driver()`. Major numbers for the pseudo-devices are assigned by `install_driver()`. Character major numbers and block major numbers are assigned separately for devices that are exclusively block or character. This means that two separate special device files for two different device drivers namely character and block device drivers may have the same value assigned to them as major number.

The minor number identifies a specific device, such as a single terminal. Minor numbers for devices are designated by the driver developer.

The utilities `getmajor()` and `getminor()` can be used to obtain the major and minor numbers associated with a device.

Cloning

A user process connects a stream to a driver by opening a specific device file. The connection is assigned a major number and a minor number. When a second user process (or a second open call from the same process) connects to the same driver, it will, by default, communicate with the stream head already created for the stream. But if this second user process needs to connect to a different device file under the same driver, it now becomes responsible for finding its own minor number, typically by polling for available minor device numbers under the driver. To make the task of finding minor device number easier, STREAMS supports clone opens. When a driver is implemented as a cloneable device, a single node in the file system can be opened to access any unused minor devices that the driver controls, for example, the special node guarantees a separate stream for every *open* (2) system call on the driver.

In HP-UX the driver can be implemented as cloneable devices in two ways.

The first cloning method uses a special clone major number, `72`, to provide cloning. For each cloneable device, a device file must exist that has the clone major number of `72` and also has a minor number equal to the major number of the real device. When an application opens this type of device file, STREAMS passes the driver open routine `CLONEOPEN` in the *`sflag`* parameter. The driver allocates a minor number and returns a new device number containing the true major number and the chosen minor number.

The second cloning method is useful for drivers which need to be able to encode information in their minor numbers. This is not possible in the first method, as the clone device file for that method must have as its minor number the major number of the driver being cloned.

In the second cloning method, the driver designates a particular minor number as its “clone” minor number. The driver open routine checks the minor number portion of the device number parameter passed to it, and if it is the clone minor number, the driver open routine allocates a minor number and returns a new device number to the caller, in the same way as the first cloning method described. The returned device number must contain both a major number and the new minor number. A driver using this cloning method may also change the major number in the device number it returns. However, the new major number must correspond to a STREAMS driver with the same `streamtab` structure as the driver associated with the original major number.

Flow Control in Drivers

In general, the same utilities and mechanisms used in implementing module flow control are used by drivers too.

STREAMS allows flow control to be used on the driver read-side to handle temporary upstream blocks. The driver or a module has an option of resetting the stream head read-side flow control limits by sending a `M_SETOPTS` message upstream.

Sample Driver Example

The following is an excerpt of the sample driver example:

```
/* Sample Driver inclusions */


#include <sys/types.h>

#include <sys/errno.h>

#include <sys/stream.h>

#include <sys/stropts.h>


/* Streams data structures for Drivers */


int drv_open  __((queue_t *, dev_t *, int, int, cred_t *));

int drv_close __((queue_t *, int, cred_t *));

int drv_wput  __((queue_t * q, mblk_t *));

int drv_rsrv  __((queue_t * q));


#define MOD_ID  0


static struct module_info minfo =  {

        MOD_ID, "drv", 0, INFPSZ, 65536, 1024

};


static struct qinit drv_rinit = {

        NULL, drv_rsrv, drv_open, drv_close, NULL, &minfo

};


static struct qinit drv_winit = {

        drv_wput, NULL, NULL, NULL, NULL, &minfo

};


static drv_info_t drv_info = {

      "drv",                              /* Driver name */

      "pseudo",                           /* Driver Class */
      DRV_CHAR | DRV_PSEUDO | DRV_MP_SAFE, /* Driver flags */
      -1,                                  /* Block Major number */
      -1,                                  /* Character Major number */
      NULL, NULL, NULL                     /* cdio, gio_private, cdio_private */
}

static drv_ops_t test_drv_ops = {
    NULL,                   /* d_open     */
    NULL,                   /* d_close    */
    NULL,                   /* d_strategy */
    NULL,                   /* d_dump     */
    NULL,                   /* d_psize    */
    NULL,                   /* d_mount    */
    NULL,                   /* d_read     */
    NULL,                   /* d_write    */
    NULL,                   /* d_ioctl    */
    NULL,                   /* d_select   */
    NULL,                   /* d_option1  */
    NULL, NULL, NULL, NULL, /* reserved entry points */
    NULL                    /* d_flags    */
};

static streams_info_t drv_str_info = {
      "drv",                                    /* Module name */
      -1,                                       /* major no */
      { &drv_rinit, &drv_winit, NULL, NULL },   /* streamtab */
      STR_IS_DEVICE | MGR_IS_MP | STR_SYSV4_OPEN,  /* streams flags */
      SQLVL_QUEUEPAIR,                          /* sync level */
      "",                                       /* elsewhere sync name */
};

struct streamtab drvinfo = {
       &drv_rinit,
       &drv_winit
};
```

NOTE Unlike a module, when a value of -1 is specified as a major number in a driver's `streams_info_t struct`, STREAMS assigns a unique major number when the driver gets installed on a system.

Installation Routine

Like STREAMS modules, STREAMS drivers need to configure the STREAMS specific information by calling `str_install()` with a defined `streams_info_t`. In addition, since drivers do have `cdevsw` information, a call to `install_driver()` should be made before the configuration for streams is done in `str_install()`.

```
int
drv_install()
{
       int retval;

       if ((retval = (install_driver(&drv_info, &drv_ops))) !=0)
           return retval;

       if ((retval = str_install(&drv_str_info)) !=0) {
   (void)uninstall_driver(&drv_info);
           return retval;
       }

       return 0;
}
```

Open/Close Routines

At the system interface, the hardware device drivers and the character I/O drivers generally have direct entry points to process `open`, `close`, `read`, `write`, IOCTL and select calls. For the STREAMS drivers however these entry points are accessed via STREAMS and their format differs from the non-STREAMS character device drivers (`drv_ops_t` for a STREAMS driver is usually NULL).

The STREAMS mechanism allows only one stream per minor device. The driver open is called whenever a STREAMS device is opened.

```
int
drv_open(q, dev, oflag, sflag, credp)
queue_t *q;
dev_t   *dev;
int     oflag;
int     sflag;
cred_t  *credp;
{

    drv_priv_t *drvp;  /* Pointer to the module's private data */
    dev_t      dev_minor;

    /*
     * Assign a minor number depending on whether this is a clone or
     * a non-clone open
     */
    if (sflag != CLONEOPEN) {
        /* Non-clone open */
        ...
    } else {
        /* Clone open */
        ...
        *dev = makedev(major(*dev), dev_minor);
    }

    /*
     * Allocate the driver specific private data to be assigned to
     * the q_ptr (per-stream data)
     */
    drvp = (drv_priv_t *)kmem_alloc(sizeof(drv_priv_t), M_NOWAIT);
    if (!drvp) {
        return 1 ; /* Failure */
    }

    /*
     * Assign the private data structure to both the read and the write
     * side q_ptr's.
     */
    q->q_ptr = WR(q)->q_ptr = drvp;

    ...
    return 0 ; /* Success */
}

int
drv_close(q, cflag, credp)
queue_t *q;
int     cflag;
cred_t  *credp
{

     flushq(WQ(q), FLUSHALL);
     /*
      * Free any module specific resources allocated during open for
      * this instance of the module.
      */
     if (q->q_ptr)
         kmem_free((caddr_t)q->q_ptr, sizeof(drv_priv_t));
     /*
      * Free the minor number allocated for this stream
      */
     ...
     /*
      * Assign NULL to both the read and the write side q_ptr's
      */
     q->q_ptr = WR(q)->q_ptr = NULL;

     return 0 ; /* Success */
}
```

Put Procedure

The put procedure is the third entry point for the STREAMS driver.

Outbound Processing

The write (2) and *ioctl* (2) system calls are only seen by the stream head. The stream head translates the *write* (2) and *ioctl* (2) system calls into messages and sends them downstream to be processed by the drivers's write-side put procedure. If the message cannot be sent immediately to the hardware or the software device, it may be stored on a driver's write queue if a write-side service procedure exists else may be in the driver's private data structures.

An example of a driver's write-side put procedure is shown:

```
int
drv_wput(q, mp)
queue_t *q;
mblk_t  *mp;
{

     ...
     if ((mp->b_datap->db_type >= QPCTL) &&
  (mp->b_datap->db_type != M_FLUSH)) {
         /*
          * Process the high priority message
          */
         ...
         return;
     }
```

```
switch (mp->b_datap->db_type) {
    case M_DATA:
    case M_PROTO:
         /* process the message */
         ...
         ...
    case M_IOCTL:
         struct iocblk * iocp = (struct iocblk *)mp->b_rptr;

         switch (iocp->ioc_cmd) {
             /* ioctl commands recognized by the driver */
             case D_IOCTL1:
                  /*
                   * process and reply back with a positive
                   * or negative acknowledgment
                   */
                  ...
                  if (success) {
                      ...
                      mp->b_datap->db_type = M_IOCACK;
                  } else {
                      ...
                      mp->b_datap->db_type = M_IOCNAK;
                  }
                  qreply(q, mp);
                  return;
             case D_IOCTL2:
                  ...
                  ...
             /* ioctl commands not recognized by the driver */
             default:
                  /* Send a negative acknowledgement upstream */
                  mp->b_datap->db_type = M_IOCNAK;
                  qreply(q, mp);
                  return;
         }
    case M_FLUSH:
         if (*mp->b_rptr & FLUSHW) {
             /* Flush the write queue */
             ...
```

```
              }

              if (*mp->b_rptr & FLUSHR) {
                  *mp->b_rptr &= ~FLUSHW;
                  /* Flush the read queue if messages can be enqueued */
                  ...
                  qreply(q, mp);
              } else
                  freemsg(mp);
              return;
         default:
              /* Free the message if the driver does not
               * understand it.
               */
              freemsg(mp);
              return;
     }
}
```

Inbound Processing

The *read* (2)system call is only seen by the stream head, which processes the system call. A STREAMS driver will not be aware of this system call. When the driver is ready to send data upstream, it builds an appropriate message and sends it to the read queue of the appropriate stream. The driver's interrupt routine usually does a `putnext()` on the driver's read-side queue to send the message upstream. The message to be sent upstream can be enqueued on the driver's read-side queue, to be processed by the service procedure either to honor the flow control mechanisms in STREAMS or to reduce the amount of time spent by the interrupt routine in processing these messages.

A driver could optionally implement flow control and put the message on it's own queue to be processed later by the service procedure.

NOTE A driver may or may not define the read-side put and service procedures. This exception of not having a read-side put procedure is only applicable to a driver.

Service Procedure

Services procedures are optional for the drivers too and defined when the messages can get enqueued on to a driver's read-side queue.

The following example shows a read service procedure for a driver:

```
int
drv_rsrv(q)
queue_t *q;
{

    mblk_t *mp;

    while ((mp = getq(q)) != NULL) {
        /* check for flow control */
        if (!(mp->b_datap->db_type >= QPCTL) &&
              !canputnext(q) {
            putbq(q,mp);
            return;
        }

        /* Process the message */
        switch (mp->b_datap->db_type) {
            case M_DATA:
                 ...
                 ...
        }
     } /* while */
}
```

Driver Specific Design Guidelines

In addition to the guidelines listed in the *"Design Guidelines"* section, the driver developers should follow these guidelines:

- A driver must be defined and configured in a kernel before it can be opened.
- Drivers managing a hardware device must have an interrupt service procedure.
- Messages that are not understood by the driver should be freed.
- `The M_IOCTL` messages must be processed and acknowledged by the driver, or the stream head will block for an `M_IOCACK` or `M_IOCNAK` until the timeout (which could potentially be infinite) expires.
- If a driver does not understand the IOCTL, it must send an `M_IOCNAK` upstream.
- Drivers are responsible for processing all `M_FLUSH` messages and whenever appropriate turn `M_FLUSH` messages around, i.e., sending the `M_FLUSH` message upstream.
- Driver developers should be very careful with the `M_ERROR` messages, as an error message `M_ERROR` received at the stream head could lock up the stream.
- If a driver wants to allocate a controlling terminal, it should send a `M_SETOPTS` message with the `SO_ISTTY` flag set upstream.

DLKM STREAMS

Traditionally, kernel modules have been statically bound into the main kernel executable, `/stand/vmunix`. This method requires system administrators to rebuild the kernel executables and then reboot the system in order to **add**, **remove**, or **patch** a kernel module.

A **Dynamically Loadable Kernel Module** (DLKM) can be loaded into the running kernel without the need for a rebuild or a reboot. For a module to be dynamically loadable, it must have a few extra data structures and functions that are not needed for static modules. The details of the DLKM related data structures, functions and how to construct a DLKM STREAMS module/driver are documented in the *HP-UX Driver Development Guide*.

The *HP-UX Driver Development Guide* is available for free download at `http://www.hp.com/go/hpux_ddk`.

5 Multiplexing

This chapter describes the STREAMS multiplexing feature of the STREAMS framework. It provides design guidelines on building, dismantling, and configuring multiplexors.

Overview

STREAMS multiplexing is a special feature in the STREAMS framework. It provides a mechanism to connect multiple streams below a driver so that data can be routed among the connected streams. The special purpose STREAMS driver used in implementing STREAMS multiplexing is called a STREAMS multiplexor.

A multiplexor is logically partitioned into an upper-half and a lower-half. The upper-half deals with the streams opened to the multiplexor while the lower-half deals with the streams linked under the multiplexor.The upper-half of a multiplexor acts like a software driver. It follows the same rules regarding unrecognized messages, flushing, and M_IOCTL processing. The lower-half of a multiplexor acts like a stream head when processing messages.

A multiplexor which multiplexes data from several upper stream to a single lower stream is called an N-to-1 or upper multiplexor. A multiplexor that has only one upper stream but several lower streams is called a 1-to-M or lower multiplexor. M-to-N multiplexing configurations are implemented by using both mechanisms in a driver. In addition to M-to-N multiplexors configuration, more complex configurations can be created by connecting streams containing multiplexors to other multiplexors.

The M-to-N configuration is useful in implementing protocols which route data between multiple upper and lower streams. For example, the IP multiplexing driver is an M-to-N configuration.

Multiplexor configurations can be built and dismantled at the user level by using `I_LINK/I_UNLINK ioctl` commands.

A single stream can be linked under one multiplexor only. The number of streams that can be linked to a multiplexor is implementation dependent and is not controlled by STREAMS framework.

The relationship between the multiple streams connected above or below the multiplexor is opaque to STREAMS framework. It is the responsibility of the STREAMS multiplexor to route data between the appropriate streams and to handle the flow control condition. STREAMS does not directly support the flow control between multiplexed streams. The flow control in a multiplexor is discussed in “Flow Control in a Multiplexor” on page 136

Building and Dismantling Multiplexors

Multiplexor configurations can be built and dismantled at the user level by using `I_LINK/I_UNLINK ioctl` commands.

To Build a Multiplexor

Multiplexor configurations are created at the user level via system calls. The multiplexor driver is like any other software driver. It owns a node in the file system and is opened just like any other STREAMS device driver.

The `mux` multiplexor, illustrated in Figure 5-1, multiplexes stream(s) opened to it over a single lower stream to the `drv` driver.

Opening the `mux` multiplexor and the `drv` device driver creates two distinct streams as shown in Figure 5-1, “Multiplexor Before Link,”. The `drv` stream can be connected below the multiplexor stream using the `I_LINK ioctl` call as shown in the following code snippet:

```
muxfd = open("/dev/mux", O_RDWR);

drvfd = open("/dev/drv", O_RDWR);

muxid = ioctl(muxfd, I_LINK, drvfd);
```

`muxfd`

Is the file descriptor open to the multiplexor.

`drvfd`

Is a file descriptor of the stream to be linked under the multiplexor.

When an `I_LINK` command is executed, the stream head queues of the stream to be linked under the multiplexor are used by the multiplexor to manage its lower-half. The state of the two streams after the link is shown in Figure 5-2, “Multiplexor After Link,”.

Figure 5-1 Multiplexor Before Link

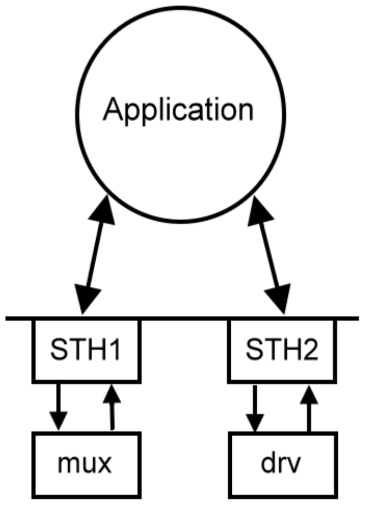
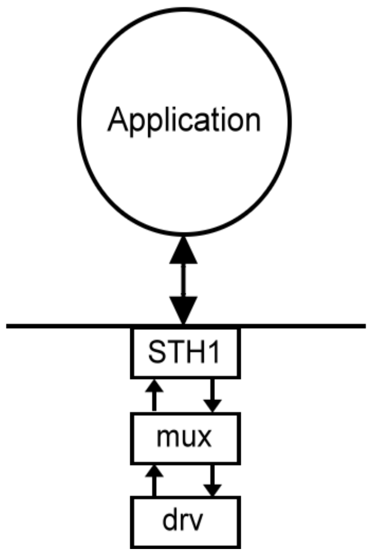


Figure 5-2 Multiplexor After Link



`I_LINK` returns an integer value `muxid` that is used by the multiplexing driver to identify the stream connected below it. This value is used for routing data through the multiplexor or dismantling the multiplexor.

The stream associated with `muxfd` is known as the controlling stream of a multiplexor link. The controlling stream is the only stream that can be used to dismantle the multiplexor that was created via the `I_LINK` command.

Once `I_LINK` is successful the file descriptor associated with the lower stream can be closed. However, this will not trigger a close as STREAMS framework maintains a reference to it. When the lower stream is unlinked, the STREAMS framework closes the stream, if there are no other references to it.

A user process cannot access streams linked below a multiplexor for the duration of the link. If the lower stream file descriptors are not closed, all future `read ()`, `write ()`, `ioctl ()`, `poll ()`, `getmsg ()`, and `putmsg ()` system calls issued to them will fail. This is because the `I_LINK` associates the stream head of each of the lower streams with the multiplexor thereby denying the user from direct access.

To Dismantle a Multiplexor

The I_UNLINK ioctl call can be used to disconnect one or all the streams connected below a multiplexor:

```
ioctl(muxfd, I_UNLINK, muxid);
```

`muxfd`

Is the file descriptor associated with the controlling stream, which was used to create the link(s). The `muxid` is associated with the stream linked below the multiplexor. If `MUXID_ALL` is specified as the third argument, then all the streams that are connected below a multiplexor are disconnected. The `MUXID_ALL` is only valid on the controlling stream that was used to create the links.

The multiplexor configuration is automatically dismantled if the `muxfd` associated with the controlling stream is closed or when the last process that references it exits. Use the persistent link mechanism if multiplexor configurations need to be retained after the controlling stream is closed. The details of persistent links are provided in the next section.

Routing Data through a Multiplexor

The criteria that is used by the multiplexors to route data between the linked streams are multiplexor implementation dependent. For example, the IP multiplexor uses the IP address specified in the protocol address to determine the subnetwork over which the data should be routed. Alternatively, the multiplexor design may require the user application to prepend data containing the muxid (returned by I_LINK) in each send request. The driver can then match the muxid in each message with the muxid of the corresponding lower stream and route the data accordingly.

Connecting and Disconnecting Multiplexor Configurations

The STREAMS framework requires two additional `qinit` structures when managing a multiplexor. These additional structures are specified in the multiplexors `streamtab` structure. A multiplexor is logically partitioned into an upper-half and a lower-half. The upper-half uses the `st_rdinit` and `st_wrinit` `qinit` structures specified in `streamtab`. The lower-half uses the `st_muxrinit` and `st_muxwinit` `qinit` structures specified in `streamtab`. The `st_muxrinit` is the lower-half read-side `qinit` structure and the `st_muxwinit` is the lower-half write-side `qinit` structure.

The multiplexor queue structures use the upper-half `qinit` structures from `streamtab`. The stream head of the stream linked below the multiplexor uses the lower-half `qinit` structures. When a stream is linked below the multiplexor, the `qinit` structures of the stream head are substituted by the lower-half `qinit` structures of the multiplexor. This linkage allows multiplexors to switch messages between upper and lower streams. When a message reaches the top of the lower stream, it is handled by the `put` and `service` procedures specified in the lower-half `qinit` structures of the multiplexor.

To Create a Multiplexor Configuration

A multiplexor configuration is created as follows:

1. An `open()` call on the multiplexing driver creates the upper stream as in any other driver. `open()` uses upper-half `qinit` structures from the multiplexor streamtab to create the driver queues. The `st_muxrinit` and `st_muxwinit` fields of the streamtab are non-null but they are ignored by the `open()` call. Any subsequent `open()` calls on this driver will create similar streams.
2. An `open()` on the device file will create the driver stream that we want to link under the multiplexor. The driver for this stream is typically a device driver that is compatible with the multiplexor. See Figure 5-1 on page 123.
3. Push any modules that need to be pushed on the driver stream. The stream head queues still point to its `put` and `service` procedures as specified in the stream head `streamtab`.
4. Connect the driver stream below the multiplexing stream by an I_LINK ioctl() call [See streamio(7)]. The configuration now looks like Figure 5-2. The I_LINK on the multiplexor stream will modify the contents of the stream head queues of the driver stream. These contents will now point to the lower-half multiplexor's put and service procedures specified in st_muxrinit and st_muxwinit.

   During this call, the stream head of the multiplexor stream sends an `M_IOCTL` message with `ioc_cmd` set to `I_LINK` to the multiplexing driver. The `M_DATA` part of the `M_IOCTL` contains an `linkblk` structure. The multiplexing driver stores the `linkblk` structure information in its private storage and returns an `M_IOCACK` message. The `l_index` is returned to the user space process as `muxid` to request an `I_UNLINK` later.

   The `linkblk` structure contains following fields:

   `l_qtop`

   Is the multiplexor's write queue.

   `l_qbot`

   Is the stream head write queue of the stream linked below multiplexor.

   `l_index`

   Is a unique (system wide) identifier for the link.

The above plumbing causes any messages reaching the top of the stream from the driver below to be delivered to the multiplexing driver's lower-half read-side `put` and `service` procedures. The multiplexing stream becomes the controlling stream. This plumbing relationship is remembered by the STREAMS framework until the controlling stream is closed, or the stream below is unlinked through `I_UNLINK`.

To Disconnect a Multiplexor Configuration

Disconnection of multiplexor configurations involves unlinking lower streams. The unlinking can be accomplished by the following three methods:

- An `I_UNLINK ioctl` referencing a specific stream.
- An `I_UNLINK ioctl` indicating all lower streams.
- The last `close()` on the upper controlling stream.

An `I_UNLINK ioctl` performs the unlink operation as follows:

1. The controlling stream receives a `muxid` or `MUXID_ALL` value in the `I_UNLINK ioctl`. This specifies an unlink operation on an individual link or an unlink operation on all the lower streams respectively.
2. The controlling stream remembers all the link operations processed through its stream head. It sends an `linkblk` structure down to the multiplexing driver through an `M_IOCTL` message for each unlink operation. If the user specified `MUXID_ALL` in the `I_UNLINK ioctl`, the multiplexor sees a series of individual unlinks. If the user specified a muxid that is returned from the unlink operation, a single unlink request with the muxid in the l_index is sent to the multiplexing driver in the `M_IOCTL` message.
3. The controlling stream will replace the lower stream head queue's `qinit` structures with the original stream head queue's `qinit` structures.

NOTE The `I_UNLINK ioctl` performs the unlink operation as follows.

- If no reference exists for a lower stream (i.e. the lower stream has been closed), a subsequent unlink will automatically close the stream. Otherwise, the lower stream must be closed after the unlink operation.
- STREAMS will automatically dismantle all cascaded multiplexors if their controlling stream is closed.
- An `I_UNLINK` will leave lower, cascaded multiplexing streams intact unless the file descriptors of the cascaded multiplexing streams were previously closed.

Characteristics of Multiplexing Configurations

In a multiplexing configuration, streams opened to the multiplexor are referred to as upper streams and those linked below the multiplexor are referred to as lower streams.

- An `I_LINK` is required for each lower stream connected to the driver. Additional upper streams can be connected to the multiplexing driver by `open()` calls.
- An upper stream provides the only interface between the user processes and the lower stream(s). The lower stream(s) are not accessible from the user space.
- System calls (except `close()`) on the lower stream return `EINVAL`. Therefore, all modules that need to be pushed on the lower stream need to be pushed before an `I_LINK` operation.
- No direct linakge is established between the upper and lower streams.
- Messages flowing upstream from the driver enter the multiplexing driver's read side put procedure. The multiplexing driver then has to route the messages to the appropriate upper or lower stream(s). Similarly, messages flowing downstream from user space or any upper stream(s) to the multiplexing driver have to be to be processed and routed by the multiplexing driver.
- STREAMS flow-control needs to be handled by the multiplexing driver as there is no direct linkage between upper and lower streams.
- Multiplexing drivers must be implemented so that new streams can be dynamically connected to (and existing streams disconnected from) the driver without interfering with its operation. The number of streams that can be connected to a multiplexor is implementation dependent.

Persistent Links

When I_LINK and I_UNLINK ioctls are used, the file-descriptors associated with the upper stream need to be active through out the operation of the multiplexor configuration. Closing the controlling stream will unplumb the multiplexor configuration. It may not be desirable to keep an application process active only to hold the multiplexor configuration together. This is resolved by using persistent links below a multiplexor. A persistent link is similar to a STREAMS multiplexor link, except that a process is not needed to maintain the links. After the multiplexor configuration has been set up, the process may close all associated file descriptors and exit. The multiplexor will remain intact.

Persistent links are created and dismantled with two ioctls:

- I_PLINK
- I_PUNLINK

close(2) and I_UNLINK cannot disconnect a multiplexor configuration created through the I_PLINK ioctl.

Creating Persistent Links

The format of the I_PLINK is:

```
ioctl(muxfd, I_PLINK, drvfd);
```

muxfd

Is the stream connected to the multiplexing driver.

drvfd

Is the stream to be linked below the multiplexing driver.

The ioctl I_PLINK returns the muxid of the configuration. The muxid can be stored in a file or passed to another process for unplumbing the persistently linked streams later.

The following code snippet will create the persistent link configuration.

```
...
muxfd = open("/dev/mux", O_RDWR);
drvfd = open("/dev/drv", O_RDWR);
muxid = ioctl(muxfd, I_PLINK, drvfd);
/* Save the muxid in file for later use */
...
```

Figure 5-3, “Multiplexor Before I_PLINK,” shows how open() creates a stream between the device and the stream head. Figure 5.4 shows a multiplexor after I_PLINK.

Figure 5-3 Multiplexor Before I_PLINK

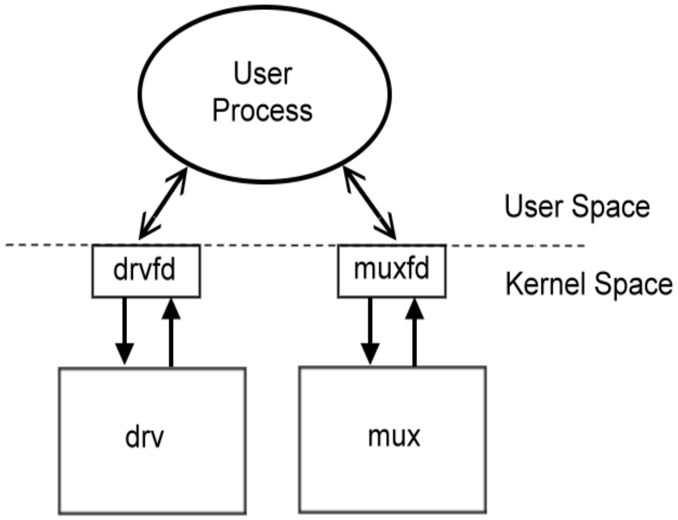
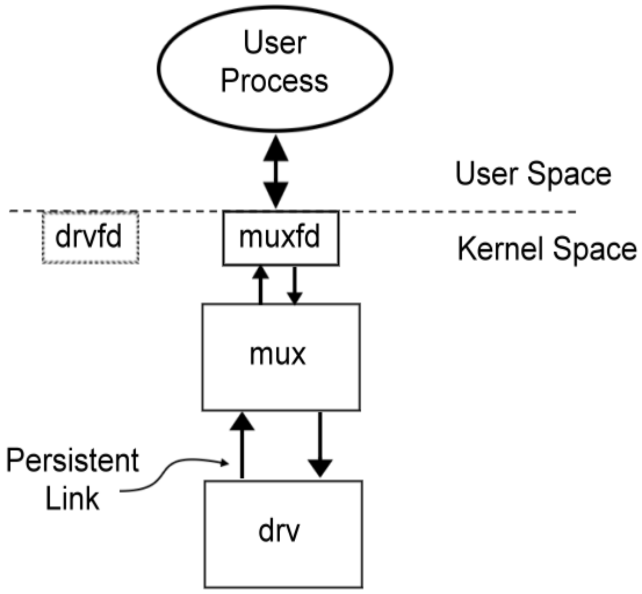
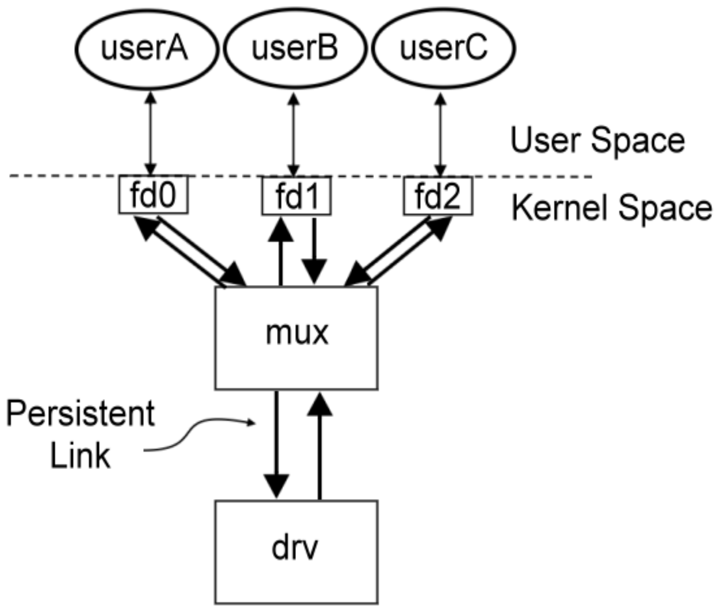


Figure 5-4 Multiplexor After I_PLINK



Once the persistent link is setup, users can open the `mux` multiplexor and send the data to the `drv` driver as the persistent link to the `drv` driver remains intact. See Fig. 5.5

Figure 5-5 Data Transfer to the Driver



Dismantling Persistent Links

Use `ioctl I_PUNLINK` to dismantle the persistent link between a stream opened to the multiplexor and the stream connected below the multiplexing driver. Its format is:

```
ioctl(muxfd, I_PUNLINK, muxid);
```

`muxfd`

Is a file descriptor associated with a stream opened to a multiplexing driver (upper stream).

`muxid`

Is the muxid returned by the `I_PLINK ioctl` for the stream that was connected below the multiplexor.

There may be multiple streams persistently linked beneath a multiplexing driver. Each of these persistent links can be individually unlinked with the `I_PUNLINK ioctl` by specifying the `muxid` associated with them, or an `I_PUNLINK ioctl` with the `muxid` value of `MUXID_ALL`.

The following code snippet will dismantle the configuration given in Figure 5.4.

```
...
muxfd = open("/dev/mux", O_RDWR);
   /* retrive the saved muxid */
ioctl(muxfd, I_PUNLINK, muxid);
...
```

Characteristics of Persistent Links

Regular links created through `I_LINK` cannot be unlinked with `I_PUNLINK`. Similarly, persistent links created through `I_PLINK` cannot be unlinked through `I_UNLINK`. The `ioctl` will return `EINVAL` when the `I_LINK`/`I_UNLINK` and `I_PLINK`/`I_PUNLINK` calls are intermixed.

In a multilevel configuration where persistent links exist below a multiplexor whose stream is connected to the above multiplexor by non-persistent links, closing the controlling stream will remove the non-persistent links but persistent links will stay intact.

In a multilevel configuration where a non-persistent link exists below a multiplexor whose stream is connected to the above multiplexor by persistent links, removing the persistent links will remove the non-persistent links, if no other references to the lower streams exist.

STREAMS Multiplexor

A STREAMS multiplexing driver or multiplexor is a special purpose STREAMS driver that provides a way to route messages between different streams. Various multiplexor configurations can be built and dismantled as explained in the previous sections.

The upper-half of the multiplexor:

- Deals with the streams opened to the multiplexor.
- Follows the same rules as the streams drivers do regarding unrecognized messages, flushing, and `M_IOCTL` processing.

The lower-half of the multiplexor:

- Deals with the streams linked below the multiplexor.
- Acts like a stream head when processing messages.

A STREAMS multiplexor differs from the regular STREAMS driver in the following ways:

- A multiplexor routes messages between multiple streams, instead of between an I/O device and the streams connected to the device.
- For multiplexing drivers, the first two fields in the `streamtab` structure, `st_rdinit` and `st_wrinit` define the read-side and the write-side `qinit` structures of the upper-half of the multiplexor. The other two entries, `st_muxrinit` and `st_muxwinit`, define the read-side and the write-side `qinit` structures of the lower-half of the multiplexor.

Ioctl Processing in a Multiplexor

For M_IOCTL processing, the upper-half of the multiplexor must follow the same rules as drivers. It must always process an M_IOCTL message from the stream head and acknowledge it with a M_IOCACK/M_IOCNAK. If the upper-half of the multiplexor does not understand the ioctl, it must send an M_IOCNAK upstream.

In addition, all STREAMS multiplexing drivers must handle the ioctls I_LINK/I_PLINK and I_UNLINK/I_PUNLINK messages so that they can build and dismantle a multiplexing configuration.

Ioctl processing is not applicable to the lower-half of the multiplexor. This is because a stream linked under a multiplexor will not be accessible to anything other than that multiplexor.

Flush Handling in a Multiplexor

On receiving an M_FLUSH message, the upper-half of the multiplexor behaves like a driver and the lower-half of the multiplexor behaves like a stream head.

On receiving an M_FLUSH message from a stream linked underneath it at a lower read queue, the lower-half of the multiplexor will flush the lower read queue, if FLUSHR is set, and unsets FLUSHR. If FLUSHW is not set, it frees the message. Otherwise, it flushes the corresponding lower write queue and sends the M_FLUSH message downstream.

To handle M_FLUSH messages received at a upper write queue, the upper half of the multiplexor flushes the upper write queue if FLUSHW is set and unsets FLUSHW flag. If FLUSHR is not set, it frees the message. Otherwise, it flushes the corresponding upper read queue and sends the M_FLUSH message upstream.

Flow Control in a Multiplexor

When multiple streams are feeding a stream, or a single stream is feeding multiple streams in a multiplexor configuration, one or more of the receiving streams can be flow controlled. If the multiplexor implements flow control, the sending streams should enqueue the messages on their own queues and process them when they are back-enabled.

Unlike the rest of the stream, the queues in the upper and the lower-half of the multiplexor are not linked together through q_next pointers. The absence of the direct linkage between the upper and lower streams means that the flow control has to be handled by special code in the multiplexing driver. To implement flow control, multiplexors must have a lower write service procedure and an upper read service procedure.

Typically, the multiplexor's lower write service procedure will be enabled when the write side of a stream linked under a multiplexor has its flow control lifted. The lower write service procedure must then enable all the upper write queues that were blocked from sending messages downstream when the lower stream was flow controlled.

The multiplexor's upper read service procedure is enabled when the read side of the multiplexor stream has its flow control lifted. The upper read service procedure must then enable all the lower read queues that were blocked from sending messages upstream to the stream that was flow controlled.

The decision to block all the sending streams when one or more receiving streams are flow controlled is multiplexor-implementation specific.

A Sample Multiplexing Driver

This section contains an example of a STREAMS multiplexing driver that implements an N-to-1 configuration.

Multiplexor Declaration

```
/* Sample Multiplexor inclusions */


#include <sys/types.h>

#include <sys/errno.h>

#include <sys/stream.h>

#include <sys/stropts.h>


/* data structures for a STREAMS multiplexing driver */


int mux_open   __((queue_t *, dev_t *, int, int, cred_t *));

int mux_close  __((queue_t *, int, cred_t *));


/* Upper read procedures */

int mux_ursrv  __((queue_t * q));


/* Upper write procedures */

int mux_uwput  __((queue_t * q, mblk_t *));

int mux_uwsrv  __((queue_t * q));


/* lower read procedures */

int mux_lrput  __((queue_t * q, mblk_t *));

int mux_lrsrv  __((queue_t * q));


/* lower write procedures */

int mux_lwsrv  __((queue_t * q));


#define MOD_ID  0


static struct module_info minfo =  {

        MOD_ID, "MUX", 0, INFPSZ, 65536, 1024

};


static struct qinit mux_urinit = {   /* upper read */

        NULL, mux_ursrv, mux_open, mux_close, NULL, &minfo

};
```

```
static struct qinit mux_uwinit = {  /* upper write */
        mux_uwput, mux_uwsrv, NULL, NULL, NULL, &minfo
};

static struct qinit mux_lrinit = {  /* lower read */
        mux_lrput, mux_lrsrv, NULL, NULL, NULL, &minfo
};

static struct qinit mux_lwinit = {  /* lower write */
        NULL, mux_lwsrv, NULL, NULL, NULL, &minfo
};

static drv_info_t mux_info = {
      "MUX",                            /* Driver name */
      "pseudo",                         /* Driver Class */
      DRV_CHAR | DRV_PSEUDO | DRV_MP_SAFE, /* Driver flags */
      -1,                               /* Block Major number */
      -1,                               /* Character Major number */
      NULL, NULL, NULL                  /* cdio, gio_private, cdio_private */
}

static drv_ops_t mux_drv_ops = {
    NULL,                   /* d_open     */
    NULL,                   /* d_close    */
    NULL,                   /* d_strategy */
    NULL,                   /* d_dump     */
    NULL,                   /* d_psize    */
    NULL,                   /* d_mount    */
    NULL,                   /* d_read     */
    NULL,                   /* d_write    */
    NULL,                   /* d_ioctl    */
    NULL,                   /* d_select   */
    NULL,                   /* d_option1  */
    NULL, NULL, NULL, NULL, /* reserved entry points */
    NULL                    /* d_flags    */
};

static streams_info_t mux_str_info = {
      "MUX",                                    /* Module name */
      -1,                                         /* major no */
      { &mux_urinit, &mux_uwinit, &mux_lrinit, &mux_lwinit}, /* streamtab */
      STR_IS_DEVICE | MGR_IS_MP | STR_SYSV4_OPEN,  /* streams flags */
      SQLVL_QUEUEPAIR,                             /* sync level */
      "",                                          /* elsewhere sync name */
};

struct streamtab mux_strtab = {
       &mux_urinit,
       &mux_uwinit,
       &mux_lrinit,
       &mux_lwinit
};

/* data structures private to the multiplexor used in this example */

/* Number of upper streams allowed in this N-to-1 configuration */
#define MUXCNT 10

struct mux{
    queue_t *top_rq;  /* pointer to the multiplexor's upper read queue */
}mux_data_t;

mux_data_t mux_list[MUXCNT];

queue_t *muxbot_wq;  /* write-q of the linked lower stream */
int muxerr;  /* Set if the lower stream recv's a M_ERROR/M_HANGUP */
```

Installation Routine

Like other STREAMS drivers, STREAMS multiplexing drivers also need to configure the STREAMS specific information by calling `str_install()` with a defined `streams_info_t`.

```
int
mux_install()
{
   int retval;

   if ((retval = (install_driver(&mux_info, &mux_drv_ops))) !=0)
       return retval;

   if ((retval = str_install(&mux_str_info)) !=0)
{
       (void)uninstall_driver(&mux_info);
       return retval;
   }

   return 0;
}
```

Open and Close Routines

The upper read queue contains the canonical driver `open` code. Each successful open of the multiplexor updates the `mux` data structures in the `mux_list`, and assigns this address to the `q_ptr` of the upper read and write queues. The minor number associated with the multiplexing driver is used as an index for `mux_list`.

The `close` routine below clears the `mux` entry set during the `open`, so that this queue will no longer be found.

```
int
mux_open(q, dev, oflag, sflag, credp)
queue_t *q;    /* multiplexor's upper read queue */
dev_t   *dev;
int     oflag;
int     sflag;
cred_t  *credp;
{

    mux_data_t *muxp;  /* Pointer to the multiplexor's private data */
    int        index;
    int        minor_dev;

    /*
     * Assign a minor number depending on whether this is a clone or
     * a non-clone open and make sure not to open more than MUXCNT streams.
     */
    if (sflag != CLONEOPEN)
 {
        /* Non-clone open */
        minor_dev = minor(*dev);
    } else {
        /* Clone open */
        for(minor_dev=0; minor_dev < MUXCNT; minor_dev++)
            if (mux_list[minor_dev].top_rq == NULL) break;
    }
    if (minor_dev >= MUXCNT) return ENOENT; /* Failure */

    muxp = &mux_list[minor_dev];
    muxp->top_rq = q;
    /*
     * Assign the private data structure to both the read and the write
     * side q_ptr's.
     */
    q->q_ptr = WR(q)->q_ptr = muxp;

    return 0 ; /* Success */
}

int
mux_close(q, cflag, credp)
queue_t *q;
int     cflag;
cred_t  *credp
{
     ((mux_data_t *)q->q_ptr)->top_rq = NULL;
     /*
      * Assign NULL to both the read and the write side q_ptr's
      */
     q->q_ptr = WR(q)->q_ptr = NULL;

     return 0 ; /* Success */
}
```

Put Procedure

In a multiplexing configuration:

- The upper write put and the lower read put procedures are mandatory.
- The upper read put and the lower write put procedures can be skipped over and the message put to the next queue if they are not mandatory.

In this example, the upper read side put procedure is not used. The lower stream read queue procedures transfer the messages directly to the read queue upstream from the multiplexor. There is no lower write put procedure either. This is because the upper write procedures directly feed the lower write queue downstream from the multiplexor.

Upper Write Put Procedure

Downstream data written to an upper stream in a multiplexing configuration is queued on the corresponding upper write queue if the lower stream is flow controlled.

The upper write put procedure does the following:

- Trap all ioctls and in particular handle the I_LINK/I_PLINK and I_UNLINK/I_PUNLINK ioctls.
- Handle M_FLUSH message like a driver.
- Discard any unrecognized messages it receives.

An example of a upper write put procedure is shown below:

```
int
mux_uwput(q, mp)
queue_t *q;
mblk_t  *mp;
{
    ...
    if ((mp->b_datap->db_type >= QPCTL) &&
        (mp->b_datap->db_type != M_FLUSH)) {
        /* Process the high priority message */
        return;
    }

    switch (mp->b_datap->db_type) {
        case M_DATA:
        case M_PROTO:
            /* Let the service procedure do the processing */
            putq(q,mp);
            break;
        case M_IOCTL:
            struct iocblk * iocp = (struct iocblk *)mp->b_rptr;
            switch (iocp->ioc_cmd) {
                /* plumb/unplumb ioctls */
```

```
case I_LINK:
case I_PLINK:
    if (muxbot_wq != NULL)
        /* lower stream is already linked */
        mp->b_datap->db_type = M_IOCNAK;
    else {
        /* Obtain the linkblk from the message
         * and set up the link
         */
        linkp = (struct linkblk *)mp->b_cont->b_rptr;
        muxbot_wq = linkp->l_qbot;
        muxerr = 0;
        iocp->ioc_count = 0;
        mp->b_datap->db_type = M_IOCACK;
    }
    qreply(q,mp);
    break;
case I_UNLINK:
case I_PUNLINK:
    muxbot_wq = NULL;
    mp->b_datap->db_type = M_IOCACK;
    iocp->ioc_count = 0;
    qreply(q, mp);
    break;
...
/* other ioctl commands recognized by the driver */
case D_IOCTL1:
    /* process and reply back with a positive
     * or negative acknowledgment
     */
    ...
    if (success) {
        ...
        mp->b_datap->db_type = M_IOCACK;
    } else {
        ...
        mp->b_datap->db_type = M_IOCNAK;
    }
    qreply(q, mp);
    break;
```

```
                case D_IOCTL2:
                    ...
                /* ioctl commands not recognized by the driver */
                default:
                    /* Send a negative acknowledgement upstream */
                    mp->b_datap->db_type = M_IOCNAK;
                    qreply(q, mp);
                    break;
            }
            break;
        case M_FLUSH:
            if (*mp->b_rptr & FLUSHW) {
                /* Flush the write queue */
                ...
            }
            if (*mp->b_rptr & FLUSHR) {
                *mp->b_rptr &= ~FLUSHW;
                /* Flush the read queue if messages can be enqueued */
                ...
                qreply(q, mp);
            } else
                freemsg(mp);
            break;
        default:
            freemsg(mp);
            break;
    } /* end switch */
    return;
}
```

Lower Read Put Procedure

The lower read `put` procedure receives upstream bound data from the lower stream. If the upper streams are flow controlled then messages can be queued on the lower read queue.

The lower read `put` procedure acting like a stream head should do the following:

- Handle the `M_FLUSH` messages.
- Handle the `M_ERROR` and `M_HANGUP` messages appropriately.
- Route other data messages to the upper streams based on the criteria set by the multiplexor.

An example of a lower read put procedure is shown below:

```
int
mux_lrput(q, mp)
queue_t *q;
mblk_t  *mp;
{
    ...
    switch (mp->b_datap->db_type) {
        case M_FLUSH:
            if (*mp->b_rptr & FLUSHR) {
                /* Flush the read queue */
                ...
            }
            if (*mp->b_rptr & FLUSHW) {
                *mp->b_rptr &= ~FLUSHR;
                /* flush the write queue if messages can be enqued */
                qreply(q, mp);
            } else
                freemsg(mp);
            break;
        case M_ERROR:
        case M_HANGUP:
            muxerr = 1;
            freemsg(mp);
            break;
        case M_DATA:
        case M_PROTO:
            /* let the service procedure do the processing and routing */
            putq(q,mp);
            break;
        default:
            /* Free all unrecognized messages */
            freemsg(mp);
            break;
    } /* end switch */
    return;
}
```

Service Procedure

In a multiplexing configuration, the lower write side and the upper read side service procedures are required to handle the flow control in the lower and the upper streams respectively. The upper write side and the lower read side service procedures are optional.

Upper Write Service Procedure

If there is a stream linked underneath the multiplexor, and there are no errors, `mux_uwsrv()` will take a message off the queue, process it, and send it downstream, if the lower stream is not flow controlled.

```
int
mux_uwsrv(q)
queue_t * q;
{
    ...
    if (muxerr || (muxbot_wq == NULL)) {
        /* flush all the data on this queue */
       ...
       return;
    }

    while(mp = getq(q)) {
        if (canputnext(muxbot_wq)) {
            /* Process data and send it downstream */
            ...
            putnext(muxbot_wq,mp);
        } else {
            putbq(q,mp);
            return;
        }
    }
    return;
}
```

Upper Read Service Procedure

The upper read service procedure is provided for flow control purposes only. It is scheduled to run when the flow control on the upper stream's read side is lifted. If the multiplexor still has the lower stream linked to it, `mux_ursrv()` will enable the lower streams read queue. The code for the upper read service procedure is as follows:

```
int

mux_ursrv(q)

queue_t * q;

{

   if (muxbot_wq) {

/* if the stream is still linked under the multiplexor

* enable it's read side service procedure

*/

 qenable(RD(muxbot_wq));

   return;

}
```

Lower Write Service Procedure

The lower write service procedure is responsible for enabling all the upper write queues that have messages on them. The `mux_lwsrv()` is scheduled to run when flow control on the linked stream below the multiplexor is removed. The code for the lower write procedure is as follows:

```
int

mux_lwsrv(q)

queue_t * q;

{

    ...

    for(index=0; index<MUXCNT; index++)

        if (mux_list[index].top_rq &&

             (WR(mux_list[index].top_rq))->q_first)

            qenable(WR(mux_list[index].top_rq));

    return;

}
```

Lower Read Service Procedure

The lower read service procedure is responsible for taking the messages of the read queue and routing them to the appropriate upper stream if there are no errors. It is assumed here that the messages arriving at the multiplexor's lower read queue contain the information regarding the destination stream, like a minor device number associated with the upper stream. The information is extracted and the message is appropriately routed.

The lower stream feeds the upper stream until it encounters a flow controlled stream.

```
int
mux_lrsrv(q)
queue_t * q;
{
    ...
    if (muxerr) {
        /* flush all the data on this queue */
       ...
       return;
    }

    /* Route the data to the appropriate upper stream */
    while(mp = getq(q)) {
        /* Extract the minor device number from the message */
        ...
        minor_dev = extract_minor(mp);
        /* The minor device number should be in the allowed range */
        if (minor_dev < 0 || minor_dev > MUXCNT) {
            freemsg(mp);
        } else {
            if ( mux_list[minor_dev].top_rq)
                if (canputnext(mux_list[minor_dev].top_rq))
                    putnext(mux_list[minor_dev].top_rq, mp);
                else {
                    putbq(q,mp);
                    return;
                }
        }
    } /* end while */

}
```

Multiplexor Specific Design Guidelines

The following is a list of general multiplexor design guidelines, in addition to the guidelines specified for the driver development in Chapter 4.

1. The upper-half of the multiplexor must be designed to act as the stream end, or driver, for the streams above the multiplexor.
2. The lower-half of the multiplexor must be designed to act as the stream head for the streams linked below the multiplexor.
3. The routing of message between the upper and lower-half of the multiplexor is based on multiplexor specific criteria.
4. When multiple streams are feeding a single stream, the receiving stream can get flow controlled. This possibly requires more than one feeding queue to be backenabled. When the flow control is lifted, the receiving queue's service procedure should properly backenable all the feeding queues that are holding on to messages. This is to prevent the feeding queues from being blocked indefinitely.
5. When one stream is feeding multiple streams, one of the receiving streams may get flow controlled. Depending on the functionality and the requirements of the multiplexing driver, the other streams that are not flow controlled may be starved. However, selectively forwarding the messages could result in the loss of relative time ordering of the messages.

A STREAMS IOCTL Commands

Overview

This appendix discusses the various STREAMS ioctl(s) commands.

ioctl(2) Commands

The STREAMS *ioctl* () commands are a subset of the *ioctl* (2) commands supported by HP-UX. They enable the user process to perform a variety of control functions on a stream.

Synopsis

The general syntax of an `ioctl()` directive is shown:

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

    #include <stropts.h>

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

`fildes`	An open file descriptor that refers to a stream.
`command`	Determines the control function to be performed as described.
`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.

The following IOCTL commands, with error values indicated, are applicable to all STREAMS files.

I_ATMARK

This command allows the user to see if the current message on the stream head read queue is “marked” by some module downstream. The `arg` parameter determines how the checking is done when there are multiple marked messages on the stream head read queue. It may take the following values:

ANYMARK	Checks if the message is marked.
LASTMARK	Checks if the message is the last one that is marked on the queue.
	If both ANYMARK and LASTMARK are set, ANYMARK supersedes LASTMARK.

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]`	The `arg` has an illegal value.

I_CANPUT

Checks if a certain band is writable. The `arg` parameter is set to the priority band in question. The return value is `0`, if the priority band `arg` is flow controlled. The return value is `1`, if the band is writable, or `-1` on error.

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

`[EINVAL]`	`arg` has an illegal value.

I_CKBAND

This command is used to 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. The `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]`	`arg` has an illegal value.

I_FDINSERT

This command causes the stream head to create a message from user specified buffer(s), add information about another stream and send 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:

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

```
struct strbuf      ctlbuf;

struct strbuf      databuf;

long               flags;

int                fildes;

int                offset;
```

`ctlbuf`	The `len` field in the `strbuf` structure of the `ctlbuf` field (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.
`databuf`	The `len` field in the `strbuf` structure of the `databuf` field (see *putmsg* (2)) 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`	This field specifies the type of message to be created. An ordinary (non-priority) message is created if the flags are 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 head 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 the `O_NONBLOCK` is set. Instead, it fails and sets `errno` to `EAGAIN`.
`fildes`	The `strfdinsert` structure specifies the file descriptor of the other stream.
`offset`	Specifies the number of bytes beyond the beginning of the control buffer where `I_FDINSERT` will store a pointer, must be word-aligned. This pointer will be the address of the read queue structure of the driver for the streams corresponding to `fildes` in the `strfdinsert` structure.

`I_FDINSERT` also blocks, unless prevented by the lack of internal resources, waiting for the availability of message blocks, regardless of priority or whether `O_NONBLOCK` has been specified. No partial message is sent.

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

[EINVAL]	The fildes parameter in the strfdinsert structure is an invalid open file descriptor.
[EINVAL]	The size of the pointer plus offset exceeds the value of the len field for the buffer specified through ctlptr.
[EINVAL]	Offset does not specify a properly aligned location in the data buffer.
[EINVAL]	Flags contains an undefined value.
[EFAULT]	The arg points, or ctrlbuf or databuf is outside the allocated address space.
[EAGAIN]	The IOCTL request failed because a non-priority message was to be created, the O_NONBLOCK option was set, and the stream's write queue was full because of internal flow control conditions.
[ENOSR]	Buffers could not be allocated for the message that was to be created due to insufficient STREAMS memory resources.
[ENXIO]	A hangup was received on the stream specified by fildes in the I_FDINSERT IOCTL call or on the stream specified by fildes in the strfdinsert.
[ERANGE]	The value of the len field for the buffer specified through databuf does not fall within the range for the minimum and maximum sizes of packets for the top-most module stream.
[ERANGE]	The value of the len field for the buffer specified through databuf is larger than the maximum allowable size for the data part of a message.
[ERANGE]	The value of the len field for the buffer specified through ctlbuf is larger than the maximum allowable size for the control part of a message.

The I_FDINSERT IOCTL can also fail if an error (M_ERROR) message was received by the stream specified by the fildes field in the strfdinsert structure. In this case, errno is set to the error value in the error message.

The 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_FIND

This command compares the names of all modules currently present on the stream to the name specified in arg. The command returns a value of 1, if the module is present and a value of 0 (zero) if the module is not present.

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

[EINVAL]	The arg does not contain a valid module name.
[EFAULT]	The arg points outside the allocated address space.

I_FLUSH

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

FLUSHRW	Flush write and read queues.
FLUSHW	Flush write queues.
FLUSHR	Flush read queues.

If a pipe 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 FLUSHW is set, the read queue for the other end of the pipe is flushed and the write queue for this end is flushed.

If FLUSHRW is set, the read queue of both ends of the pipe are flushed.

Correct flush handling of a pipe 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]	Could not allocate buffers for flush operation because of a lack of STREAMS memory resources.
[EINVAL]	The arg parameter is an invalid value.
[ENXIO]	A hangup was received on fildes.

I_FLUSHBAND

This command flushes a particular band of messages. The arg points to a bandinfo structure, that has the following members:

```
unsigned char     bi_pri;

int               bi_option;
```

The value of the bi_option field can be FLUSHR, FLUSHW, or FLUSHRW as described for the I_FLUSH command.

On failure, errno is set to the following value:

[EINVAL]	The bi_pr parameter value exceeds the maximum band, or the bi_option parameter is not FLUSHR, FLUSHW or FLUSHRW.

I_GETBAND

This command returns the priority band of the first message on the stream head read queue in the integer referenced by arg.

On failure, errno is set to the following value:

[ENODATA].	No message exists on the stream head read queue.

I_GETCLTIME

This command returns the close time delay in the long integer pointed to by arg.

I_SETCLTIME

This command allows the user to set the time that the stream head will delay when a stream is closing, and there is 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 is still present, data will be flushed. The arg is a pointer to the number of milliseconds to delay, rounded up to the nearest valid value on the system. The default is fifteen seconds.

On failure, errno is set to the following value:

[EINVAL]	The arg has an illegal value.

I_GETSIG

This command returns the events for which the calling process has registered to receive a SIGPOLL signal. Events are returned as in arg bitmask as defined for the I_SETSIG command.

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

[EINVAL]	User process is not registered to receive the SIGPOLL signal.
[EFAULT]	The arg points outside the allocated address space.

I_GRDOPT

This command 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]	The arg is pointing outside the allocated address space.

I_GWROPT

This command returns the current write mode setting, as described in I_SWROPT, in the int that is pointed to by the argument arg.

I_LINK

This command connects two streams:

fildes	File descriptor of the stream connected to the multiplexing driver.
arg	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 -1 on failure.

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

[EAGAIN]	Temporarily unable to allocate storage to perform the linking operation.
[EBADF]	The arg parameter, not a valid open file descriptor.
[ENXIO]	A hangup was received on fildes.
[EINVAL]	The stream referred to by fildes does not support multiplexing.
[EINVAL]	The file referred to by arg is not a stream, or the stream is already linked under a multiplexor.
[EINVAL]	The link operation would cause a “cycle” in the resulting multiplexing configuration. In other words, the driver referred to by the arg parameter is linked into this configuration at multiple places
[ENOSR]	Not enough STREAMS memory resources to allocate storage for this command.

[ETIME]	Acknowledgement message not received at stream head before timeout.

The I_LINK IOCTL can also fail if an M_ERROR or M_HANGUP message is received at the stream head for fildes before receiving the driver acknowledgement. In addition, an error can be returned in an M_IOCACK or M_IOCNAK message. When these occur, the I_LINK IOCTL fails with errno set to the value in the message.

I_LIST

This command 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 not NULL, it should point to a 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];
```

The sl_nmods indicates the number of entries the user has allocated in the array. On success, the return value is 0, sl_modlist contains the list of module names, and sl_nmods indicates the number of entries that have been filled in.

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

[EINVAL]	The sl_nmods is less than 1.
[EAGAIN]	Could not allocate buffers.

I_LOOK

This command retrieves the name of the module located just below the streams 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 FNAMESZ+1 bytes long. A #include <stropts.h> declaration is required.

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

[EINVAL]	There are no modules in the stream.
[EFAULT]	The 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, the errno is set to the following value:

[EFAULT]	The arg is pointing outside the allocated address space.

I_PEEK

Allows the user process to look (peek) at the contents of the first message on the stream head read queue. This is done without taking the message off the queue. The `I_PEEK` IOCTL operates the same way as the `getmsg()` function, except that it does not remove the message. The `arg` parameter points to a `strpeek` structure (in the `<stropts.h>` header file) with the following members:

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

The `strbuf` structure pointed to by `ctlbuf` and `databuf` has the following members:

```
int   maxlen;
int   len;
char  *buf
```

The `maxlen` field of the `strbuf` structure must specify the number of bytes of control or data information to be retrieved. The flags field can be set to `RS_HIPRI` or `0` (zero). If this field is set to `RS_HIPRI`, the `I_PEEK` IOCTL looks for a high priority message on the queue. If the field is set to `0`, the `I_PEEK` IOCTL looks at the first message on the queue.

The `I_PEEK` returns a `1` if a message was retrieved, and returns a value of `0` (zero) if no message was found; it does not wait for a message. Upon successful completion, `ctlbuf` specifies control information in the control buffer, `databuf` specifies data information in the data buffer, and flags contains `RS_HIPRI` or `0` (zero).

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

`[EINVAL]`	The flags parameter is an illegal value.
`[EFAULT]`	The `arg` points, or `ctrlbuf` or `databuf` is, outside the allocated address space.
`[EBADMSG]`	Message to be looked at is not valid for the `I_PEEK` command.

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` is connected via a persistent link below the multiplexing river. The `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 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 `-1` on failure.

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

`[ENXIO]`	A hangup was received on the stream referred to by the `fildes` parameter.
`[ETIME]`	A timeout occurred before an acknowledgement message was received at the stream head.
`[EAGAIN]`	Temporarily unable to allocate storage to perform the linking operation.
`[EBADF]`	The `arg` is not a valid open file descriptor.
`[EINVAL]`	The stream referred to by `fildes` does not support multiplexing.
`[EINVAL]`	The file referred to by `arg` is not a stream or is already linked under a multiplexing driver.
`[EINVAL]`	The link operation would cause a “cycle” in the resulting multiplexing configuration. In other words, the driver referred to by `arg` is linked into the configuration at multiple places.

The I_PLINK IOCTL can also fail if it is waiting for the multiplexing driver to acknowledge the link request and an error (M_ERROR) message, or hangup (M_HANGUP) message is received at the stream head for fildes. In addition, an error can be returned in an M_IOACK or M_IONAK message. When these occur, the I_PLINK fails with errno set to the value in the message.

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. The arg should be 0 in an I_POP request.

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

[EINVAL]	There are not modules in the stream.
[ENXIO]	Error value returned by the module being popped.
[ENXIO]	A hangup was received on fildes.

I_PUNLINK

Disconnects the two streams specified by fildes and arg that are connected with a persistent link. The fildes is the file descriptor of the stream connected to the multiplexing driver. The 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 is set to one of the following values:

[ENXIO]	A hangup was received on fildes.
[ETIME]	A timeout occurred before an acknowledgement message was received at the stream head.
[EAGAIN]	Temporarily unable to allocate storage to perform the linking operation.
[EINVAL]	The arg is an invalid multiplexor ID number.
[EINVAL]	The fildes is the file descriptor of a pipe.

An I_PUNLINK IOCTL can also fail if it is waiting for the multiplexor to acknowledge the unlink request and an error (M_ERROR) message, or hangup (M_HANGUP) is received at the stream head for fildes. In addition, an error can be returned in an M_IOCACK or M_IOCNAK message. When these occur, the P_UNLINK IOCTL fails with errno set to the value in the message.

I_PUSH

Pushes the module whose name is pointed 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 streams 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]	An invalid module name was used.
[EFAULT]	The arg points outside the allocated address space.
[ENXIO]	Error value returned by the module being pushed. The push has failed.
[ENXIO]	A hangup was received on fildes.

I_RECVFD

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

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

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

If `O_NONBLOCK` is clear, `I_RECVFD` will block until a message is present at the stream head. If `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 a `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]`	The `O_NONBLOCK` option was set, and a message was not present on the stream head read queue.
`[EFAULT]`	The `arg` parameter points outside the allocated address space.
`[EBADMSG]`	The message present on the stream head read queue did not contain a passed file descriptor.
`[EMFILE]`	Too many open files. No more file descriptors are permitted to be opened.
`[ENXIO]`	A hangup was received on `fildes`.

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.

The `I_SENDFD` command 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 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 head could not allocate a message block for the file pointer.
`[EAGAIN]`	The read queue of the receiving stream head was full and could not accept the message.
`[EBADF]`	The `arg` parameter is not a valid open file descriptor.
`[EINVAL]`	The `fildes` parameter does not refer to a stream.
`[ENXIO]`	A hangup was received on `fildes`.

I_SETSIG

Informs the stream head that the user wants the kernel to issue the SIGPOLL signal (see *signal* (2)) when a particular event has occurred on the stream associated with fildes. The 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, except where noted, of the following constants:

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.
S_ERROR	An M_ERROR message has reached the stream head.
S_HANGUP	An M_HANGUP message has reached the stream head.
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_INPUT	Any message other than an M_PCPROTO has arrived on a stream head read queue. This event is maintained for compatibility with prior releases. This is set even if the message is of zero length.
S_MSG	A STREAMS signal message that contains the SIGPOLL signal has reached the front of the stream head read queue.
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_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_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_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_WRNORM	This event is the same as S_OUTPUT.

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

Processes that want to receive SIGPOLL signals must explicitly register to receive them using I_SETSIG. If several processes register to receive the 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, the errno is set to one of the following values:

[EINVAL]	The user process is not registered to receive the SIGPOLL signal.
[EAGAIN]	A data structure to store the signal request could not be allocated.

I_SRDOPT

Sets the read mode (see *read* (2)) using the value of the argument arg. Valid arg values are:

RNORM	Byte-stream mode (default).
RMSGD	Message-discard mode.
RMSGN	Message-nondiscard mode.

Setting both `RMSGD` and `RMSGN` is an error. `RMSGD` and `RMSGN` override `NORM`.

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]`	The `arg` contains an illegal value.

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. The `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 active on a stream. Further `I_STR` calls will block until the active `I_STR` completes at the stream head. The default timeout intervals for these requests is 15 seconds. The `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`	The internal IOCTL command intended for the downstream module or driver.
`ic_timout`	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`	The number of bytes in the data argument.
`ic_dp`	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 driver in the stream can return).

The stream head will convert the information pointed to by `strioctl` structure to an internal IOCTL command message and send it downstream.

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

`[EINVAL]`	The `ic_len` field is less than `0` (zero) bytes or larger than the maximum allowable size of the data part of a message (`ic_dp`).
`[EINVAL]`	The `ic_timout` field is less than -1.

[EFAULT]	The arg points, or the buffer area specified by ic_dp or ic_len is, outside the allocated address space.
[ENOSR]	Buffers could not be allocated for the IOCTL request because of a lack of STREAMS memory resources.
[ENXIO]	A hangup was received on the stream referred to by fildes.
[ETIME]	The IOCTL request timed out before an acknowledgement was received.

The I_STR IOCTL can also fail if the stream head receives a message indicating an error (M_ERROR) or a hangup (M_HANGUP). In addition, an error can be returned in an M_IOCACK or M_IOCNAK message. In these cases, the IOCTL fails with errno set to the error value in the message.

I_SWROPT

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

SNDZERO	Sends a zero-length message downstream when a write of 0 bytes occurs. To not send a zero-length message when a write of 0 bytes occurs, this bit must not be set in arg.

On failure, the errno is set to the following value:

[EINVAL]	The arg parameter is an illegal value.

I_UNLINK

Disconnects the two streams specified by fildes and arg.

fildes	The file descriptor of the stream connected to the multiplexing driver.
arg	The multiplexor ID number that was returned by the I_LINK.

If arg is MUXID_ALL, 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, the errno is set to one of the following values:

[ENXIO]	A hangup was received on fildes.
[ETIME]	A timeout occurred before an acknowledgement message was received at the stream head.
[EINVAL]	The arg is an invalid multiplexor ID number, or fildes is already linked under a multiplexing driver.

An I_UNLINK IOCTL can also fail if it is waiting for the multiplexor to acknowledge the unlink request and an error (M_ERROR) message, or hangup (M_HANGUP) is received at the stream head for fildes. In addition, an error can be returned in M_IOCACK or M_IOCNAK message. When this occurs, the I_UNLINK IOCTL fails with errno set to the value in the message.

B STREAMS Utilities Supported by HP-UX

This Appendix deals with the STREAMS utilities supported by HP-UX. STREAMS utility routines are used to perform specific operations/functions in module and driver development. The streams utility routines are listed in alphabetical order.

NAME

adjmsg () – Trim bytes in a message.

SYNOPSIS

```
#include <sys/stream.h>

 int adjmsg (mblk_t *mp , int len);
```

PARAMETERS

mp	Pointer to a message block in a message. This block will be treated as the start of the message.
len	The number of bytes to be removed.

DESCRIPTION

`adjmsg()` trims a specific number of bytes from either the head or tail of the message pointed by *mp*. If *len* is greater than 0, `adjmsg()` trims *len* bytes from the beginning of the message starting with the first message block. If *len* is less than 0, then `adjmsg ()` removes *len* bytes from the end of the message block. No operation is performed for 0 *len* bytes.

`adjmsg()` only trims bytes across message blocks of the same message type. It fails if *mp* points to a message containing fewer than *len* bytes of the same message type starting at the message block pointed to by *mp*. The message type is determined by the type of the first message block pointed to by m*p*.

RETURN VALUES

`adjmsg()` returns 1 upon successful execution, and 0 on failure.

CONSTRAINTS

`adjmsg()` can be called from interrupt or thread context. Spinlocks can be held across a `adjmsg()` call.

NAME

allocb () – Attempts to allocate a new message block.

SYNOPSIS

```
#include <sys/stream.h>

 mblk_t *allocb (int size, int pri);
```

ARGUMENTS

size	The number of bytes in the message block.
pri	This field is no longer in use. It is provided for portability.

DESCRIPTION

The `allocb()` utility tries to allocate a message block with the requested data buffer. If the memory is not available, it will return with a NULL pointer.

RETURN VALUES

On success, `allocb()` returns a pointer to a message block of type `M_DATA` containing the data buffer of the requested size. On failure to allocate message block, it returns NULL pointer. The `bufcall()` can be used to recover from `allocb()` failures.

CONSTRAINTS

`allocb()` can be called from thread or interrupt context. Spinlocks of STREAMS/UX user lock order can be held across this call.

NAME

backq () – Returns a pointer to the queue behind a specified queue.

SYNOPSIS

```
#include <sys/stream.h>
 queue_t *backq (queue_t *q);
```

ARGUMENTS

q	Pointer to the head of the queue for a stream queue whose back queue is to be returned.

RETURN VALUES

`backq()` walks backwards from the specified *q* and returns a pointer to the queue whose `q_next` pointer is *q*. If no such queue exists, `backq()` returns NULL.

CONSTRAINTS

`backq()` can be called from thread or interrupt context. Spinlocks can be held across this call.

NAME

bcanput () – Checks for the existence of a flow control condition in the queue for a specified priority band.

SYNOPSIS

```
#include <sys/stream.h>

 int bcanput (queue_t *q, int pri);
```

ARGUMENTS

q	Pointer to the queue.
pri	Priority band for which the flow control check is requested.

DESCRIPTION

`bcanput()` routine tests if there is room left in the queue for the band specified by *pri*. If the queue does not have a service procedure, it finds the queue that contains a service procedure in the direction of message flow. It then tests to see if a message can be enqueued in the priority band specified by *pri*. If such a queue cannot be found, `bcanput()` stops at the end of the stream.

If 0 is specified in *pri*, then `bcanput()` is equivalent to `canput()`.

RETURN VALUES

Returns 1 if the message can be enqueued. If the message queue for the specified band is full, 0 is returned and `QB_WANTW` is set in the queue that has the service procedure to automatically back-enable this queue.

CONSTRAINTS

`bcanput()` can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call.

NAME

bcanputnext () – Checks for the existence of flow control in the queue for the specified priority band.

SYNOPSIS

```
#include <sys/stream.h>

 int bcanputnext (queue_t *q, int pri);
```

ARGUMENTS

q	Pointer to the queue.
pri	Priority band for which the flow control check is requested.

DESCRIPTION

`bcanputnext()` routine tests if there is room left in the queue (`q->q_next`) for the band specified in *pri*. If the queue does not have service procedure, then it finds the queue that contains service procedure in the direction of message flow and tests to see if a message priority can be enqueued. But, if such a queue cannot be found, then it stops at the end of the stream.

If 0 is specified in *pri*, then `bcanputnext()` is equivalent to `canputnext()`.

RETURN VALUES

Returns 1 if the message can be enqueued. If the message queue for the specified band is full, 0 is returned and `QB_WANTW` is set in the queue that has the service procedure to automatically back-enable the queue.

CONSTRAINTS

`bcanputnext()` can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock orders can be held across this call.

NAME

`bufcall` (), **`streams_bufcall`** () – Recover from `allocb()` failures.

SYNOPSIS

```
include <sys/stream.h>

 toid_t bufcall, (uint size, int prilev, (*func)(void *arg), void *art):
 toid_t streams_bufcall(uint size, int prilev, bufcall_fcn_t func, bufcall_arg_t arg);
```

PARAMETERS

`size`	Requested buffer size
`prilev`	It is no longer used. This parameter is provided only for portability purposes.
`func`	Address of function to be called when requested memory is available.
`arg`	Pointer to argument list to be passed to the function pointed to by *`func`*.

DESCRIPTION

This function is used when an `allocb()` request fails, and the caller wants to be notified as and when the memory becomes available. The `streams_bufcall()`/`bufcall()` will schedule the function pointed to by *`func`*, to be invoked with an argument *`arg`*, whenever memory of at least size bytes is available and return a non-zero identifier on successful scheduling. In effect, `streams_bufcall()`/`bufcall()` serves as a timeout call of indeterminate length.

The non-zero identifier returned by `streams_bufcall()`/`bufcall()` may be passed to `unbufcall()` to cancel the request.

RETURN VALUES

On success, `streams_bufcall()`/`bufcall()` returns a non-zero value that identifies the scheduling request. On failure, `streams_bufcall()`/`bufcall()` returns 0 and the function pointed to by *`func`* will not be executed.

CONSTRAINTS

`streams_bufcall()`/`bufcall()` can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock orders can be held across `streams_bufcall()`/`bufcall()`. STREAMS raises the SPL before invoking the callback *`func`* and hence the callback *`func`* should not block.

NAME

`canenable` () – Test whether queue can be enabled for running service procedure.

SYNOPSIS

```
#include <sys/stream.h>
 int canenable (queue_t *q);
```

PARAMETERS

q	Pointer to a queue.

DESCRIPTION

This function checks whether a service procedure for the specified queue can be scheduled for running.

RETURN VALUES

It returns 1 if the queue has not been disabled from running a service procedure and returns 0 if the queue is disabled.

CONSTRAINTS

`canenable()` can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call.

NAME

canput () – Test for available room in a queue.

SYNOPSIS

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

int canput (queue_t *q);
```

PARAMETERS

q	Pointer to a queue.

DESCRIPTION

`canput()` routine tests if there is room left in the queue. If the queue does not have a service procedure, then it finds the queue that contains a service procedure in the direction of message flow and tests to see if messages can be enqueued. If such a queue cannot be found it stops at the end of the stream.

RETURN VALUES

Returns 1 if the message can be enqueued. If the message queue is full, 0 is returned and `QWANTW` is set in the queue that has the service procedure, to automatically back-enable this queue.

CONSTRAINTS

`canput()` can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call.

NAME

canputnext () – Checks for the flow control conditions in a stream.

SYNOPSIS

```
#include <sys/stream.h>

 int canputnext (queue_t *q);
```

PARAMETERS

q	Pointer to a queue.

DESCRIPTION

`canputnext()` routine tests if there is room left in the queue (`q->q_next`). If the queue does not have a service procedure, then it finds the queue that contains service procedure in the direction of message flow starting at `q->q_next` and tests to see if a message priority can be enqueued. If such a queue cannot be found, then it stops at the end of the stream.

RETURN VALUES

Returns 1 if the message can be enqueued. If the message queue for the specified band is full, 0 is returned and `QWANTW` is set in the queue that has the service procedure, thereby this queue is automatically back-enabled

CONSTRAINTS

`canputnext()` can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call

NAME

cmn_err () – Display a message on the console and optionally panic the system.

SYNOPSIS

```
#include <sys/cmn_err.h>

 void cmn_err(int level, char *format...);/* args */
```

PARAMETERS

level	Severity of the error condition. Takes one of the following values:	
	CE_CONT	Used to continue a previous message or to display an informative message not connected with an error.
	CE_NOTE	To display the message preceded with NOTICE:. This is used to report the system events that do not necessarily require action, but may interest to system administrator.
	CE_WARN	To display the message preceded with WARNING:. This message is used to report the system events that require action.
	CE_PANIC	To display a message preceded with panic: and panic the system. The production code should never panic the system.
format	The message to be displayed. The message is always sent to both the system console and kernel buffer. If the first character specified in the message is an exclamation point ("!") or circumflex ("^"), cmn_err() ignores it. The functionality of cmn_err() is similar to fprintf() and the valid conversion specifications are: %c, %d, %o, %x, %s and %u.	

DESCRIPTION

The cmn_err always sends the message to both the system console and circular kernel buffer. The circular buffer is called msgbuf. The HP-UX msgbuf is a fixed size and can be viewed using the dmesg command or the kwdb debugger tool.

RETURN VALUES

None

CONSTRAINTS

cmn_err() can be called from thread or interrupt context. Spinlocks can be held across this call. If level is CE_PANIC the system will panic.

NAME

copyb () – Creates a copy of a specified message block.

SYNOPSIS

```
#include <sys/stream.h>

 mblk_t *copyb (mblk_t *mp);
```

PARAMETERS

mp	Pointer to message block from which data is to be copied.

DESCRIPTION

`copyb()` calls `allocb()` to allocate a new message. The new block will be at least as large as the block being copied. The data between the `b->rptr` and `b->wptr` in the source message block is copied into the newly allocated message block, and these pointers in the new message block are given the same offset values they had in the original message block.

RETURN VALUES

`copyb()`returns a pointer to the newly allocated message block when successful, or a NULL pointer when message block allocation fails.

CONSTRAINTS

`copyb()` can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call.

NAME

copymsg () – Makes a copy of a specified message.

SYNOPSIS

```
#include <sys/stream.h>

 mblk_t *copymsg (mblk_t *mp);
```

PARAMETERS

mp	Pointer to message to be copied.

DESCRIPTION

This function uses the `copyb()` utility to allocate and then copy message blocks from the message pointed by *mp*. It then forms the new message by appropriately linking the new message blocks together via `b_cont` pointer.

RETURN VALUES

`copymsg()` returns a pointer to the newly created message if successful, and a NULL pointer upon failure.

CONSTRAINTS

`copymsg()` can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call.

NAME

datamsg () – Checks to see if a specified message type is a data message.

SYNOPSIS

```
#include <sys/stream.h>

 int datamsg (unsigned char type);
```

PARAMETERS

type	Type of the message to be tested.

DESCRIPTION

This function is typically invoked with the message type parameter mp->b_datap->db_type, where *mp* is a message pointer. This utility checks if it is a data message type: M_DATA, M_DELAY, M_PROTO or M_PCPROTO.

RETURN VALUES

datamsg() returns 1 if the message is a data message, and 0 otherwise.

CONSTRAINTS

datamsg() can be called from thread or interrupt context. Spinlocks can be held across this call.

NAME

`drv_getparm`() – Get kernel information.

SYNOPSIS

```
#include <sys/stream.h>
 int drv_getparm (unsigned long parm, unsigned long *value_p);
```

PARAMETERS

parm	A kernel parameter of: `LBOLT`, `PPGRP`, `UPROCP`, `PPID`, `PSID`, `TIME`, or `UCRED`.
value_p	A pointer to the address to put the result field into.

DESCRIPTION

Return the value of the parameter in parm.

RETURN VALUES

`drv_getparm()` returns 0 if the `parm` is recognized or -1 if not.

CONSTRAINTS

`drv_getparm()` can be called from thread or interrupt context. Spinlocks can be held across this call.

NAME

`drv_priv` () – Get driver privilege.

SYNOPSIS

```
#include <sys/stream.h>
 int drv_priv (cred_t * cr);
```

PARAMETERS

cr	A pointer to a struct `cred_t`.

DESCRIPTION

Determine if the user has the correct credentials to perform a function.

RETURN VALUES

`drv_priv()` returns 0 if the caller has super user privilege otherwise EPERM.

CONSTRAINTS

`drv_priv()` can be called from thread or interrupt context. Spinlocks can be held across this call.

NAME

dupb () – Pointer to the message block, which is to be duplicated.

SYNOPSIS

```
#include <sys/stream.h>

 mblk_t *dupb (mblk_t *mp);
```

PARAMETERS

mp	Pointer to message block descriptor that will be duplicated.

DESCRIPTION

`dupb()` allocates a new message block structure, and copies into it the message block structure pointed by `mp`. It does not copy the data buffer. The data block pointer for the newly allocated message block is set to point to the same data block as the original message block. The reference count (`db_ref`) in the data block descriptor is incremented.

When the `db_ref` count of data block descriptor is greater than 1, then STREAMS module/driver must not modify the contents of the data buffer. If the data has to be modified, then `copyb()` should be used to create a new message block, and only the new message block should be modified.

RETURN VALUES

`dupb()` returns a pointer to the newly created message block structure if successful, and a NULL pointer upon failure.

CONSTRAINTS

`dupb()` can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call.

NAME

dupbn () – Pointer to the message block, which is to be duplicated n times.

SYNOPSIS

```
#include <sys/stream.h>

 mblk_t *dupbn (mblk_t *mp, int n);
```

PARAMETERS

mp	Pointer to message block descriptor that will be duplicated.
n	Number of duplicates.

DESCRIPTION

`dupbn()` allocates a new message block structure, and copies into it the message block structure pointed by `mp`. It does not copy the data buffer. The data block pointer for the newly allocated message block is set to point to the same data block as the original descriptor. The reference count (`db_ref`) in the data block descriptor is incremented.

When the `db_ref` count of data block descriptor is greater than 1, then STREAMS module/driver must not modify the contents of the data buffer. If the data has to be modified, then `copyb()` should be used to create a new message block, and only the new message block should be modified.

RETURN VALUES

`dupbn()` returns a pointer to the newly created message block structure if successful, and a NULL pointer upon failure.

CONSTRAINTS

`dupbn()` can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call.

NAME

dupmsg () – Duplicates a message.

SYNOPSIS

```
#include <sys/stream.h>

 mblk_t *dupmsg (mblk_t *mp);
```

PARAMETERS

mp	Pointer to a message block, which is to be duplicated.

DESCRIPTION

This function calls `dupb()` to duplicate all message block descriptors starting at *mp*, and then links the message block descriptors to form a new message. It does not copy data buffers. Since `dupb()` is used to duplicate message block descriptors, the relevant fields in the message block descriptors are initialized as described in the section on `dupb()`.

RETURN VALUES

`dupmsg()` returns a pointer to the message block on success, and a NULL pointer upon failure.

CONSTRAINT

`dupmsg()` can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call.

NAME

enableok () – Allows the service procedure for the specified queue to be scheduled.

SYNOPSIS

```
#include <sys/stream.h>
 void enableok(q);
```

PARAMETERS

q	Queue pointer.

DESCRIPTION

This function allows the service procedure for a queue to be scheduled. It does not actually schedule the service procedure. It simply undoes the effect of a previously executed `noenable()` function on this queue. If the queue was already enabled for servicing, `enableok()` has no effect.

RETURN VALUES

None

CONSTRAINTS

`enableok()` can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call.

NAME

`esballoc` () – Allocate a message block using a caller specified buffer.

SYNOPSIS

```
#include <sys/stream.h>

 mblk_t *esballoc(uchar_t *base, int size, int pri, frtn_t *fr_rtnp);
```

PARAMETERS

base	Pointer to caller data buffer.
size	Size of the data block.
pri	Priority of data block. It is no longer used. It is provided only the portability.
fr_rtnp	Pointer to `free_rtn` structure.

DESCRIPTION

`esballoc()` creates a message block, and attaches a caller-supplied data buffer to the data block and copies the contents of the `fr_rtn` structure into the message block. `esballoc()` sets `db_base`, `b_rptr`, and `b_wptr` fields to base (data buffer size) and `db_lim` to base+size. When `freeb()` is called to free the message, on the last reference to the message, the caller's free routine, specified by the `free_func` field in `fr_rtnp` structure is called with one argument, specified by the `free_arg` field, to free the data buffer.

The free routine passed to `esballoc()` can call STREAMS/UX utilities similar to the ones called by put or service procedure. Also, a free routine can safely access the same data structures as the put or service procedure of the calling module or driver. However, HP-UX does not block interrupts from all STREAMS/UX devices while the free routine runs.

RETURN VALUES

The success of `esballoc()` depends on the success of `allocb()` and that base, size, and `fr_rtn` are not NULL, in which case `esballoc()` returns a pointer to a message block. If an error occurs, `esballoc()` returns NULL.

CONSTRAINTS

`esballoc()` can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call.

NAME

`esbbcall` () – Call a function when a caller-supplied buffer can be allocated.

SYNOPSIS

```
#include <sys/stream.h>
 toid_t esbbcall(int pri, void (*bufcall_fcn_t), long bufcall_arg_t);
```

PARAMETERS

pri	It is no longer used. This parameter is provided only for portability purposes.
bufcall_fcn_t	Address of function to be called when requested memory is available.
bufcall_arg_t	Pointer to argument list to be passed to the function pointed to by `bufcall_fcn_t`.

DESCRIPTION

This function is used when an `esballoc()` request fails, and the caller notified as and when the memory becomes available. In effect, `esbbcall()` invokes the function pointed to by `bufcall_fcn_t` whenever memory becomes available.

RETURN VALUES

On success, `esbbcall()` returns a non-zero value that identifies the scheduling request. On failure, `esbbcall()` returns 0 and the caller supplied `bufcall_fcn_t` function will not be executed.

CONSTRAINTS

`esbbcall()` can be called from thread or interrupt context. Spinlocks of STREAMS/UX user lock orders can be held across `esbbcall()`. STREAMS raises the SPL level before invoking the callback function `bufcall_fcn_t` hence, the callback function should not block.

NAME

`flushband` () – Flushes the messages specified in the priority band.

SYNOPSIS

```
#include <sys/stream.h> #include <sys/stropts.h>
 void flushband (queue_t *q, unsigned char pri, int flag);
```

PARAMETERS

`q`	Queue pointer	
`pri`	Priority band to be flushed	
`flag`	Determines the type of the messages to be flushed. Valid flag values are:	
	`FLUSHDATA`	Flush data messages only (`M_DATA`, `M_PROTO`, `M_PCPROTO` and `M_DELAY`).
	`FLUSHALL`	Flush all messages.

DESCRIPTION

`flushband()` allows modules and drivers to flush messages from a specified priority band. A *`pri`* value of 0 will flush ordinary messages. Note that as a result of flushing messages, if the messages in the band fall below the band's low watermark, and if a previous module was attempting to write to this queue, then the previous service procedure will be enabled.

RETURN VALUES

None

CONSTRAINTS

`flushband()` can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call.

NAME

`flushq` () – Flushes the specified queue.

SYNOPSIS

```
#include <sys/stream.h>

 void flushq(queue_t *q, int flag);
```

PARAMETERS

q	Queue pointer.	
flag	Determines messages to be flushed. Valid flag values are:	
	`FLUSHDATA`	Flush data messages only (`M_DATA`, `M_PROTO`, `M_PCPROTO`, `M_DELAY`).
	`FLUSHALL`	Flush all messages.

DESCRIPTION

`flushq()` uses the `freemsg()` function to free each relevant message in the queue, based on the flag parameter value. If and when the queue message length falls below the low watermark and if a previous service procedure wants to write to this queue, the previous queue's service procedure is enabled.

RETURN VALUES

None

CONSTRAINTS

`flushq()` can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call.

NAME

freeb () – Deallocates a message block.

SYNOPSIS

```
#include <sys/stream.h>

 void freeb (mblk_t *mp);
```

PARAMETERS

mp	Pointer to the message block to be deallocated.

DESCRIPTION

`freeb()` deallocates a message block, and if the `db_ref` count of the `datab` structure it is pointing to is greater than 1, it decrements the `db_ref` count by 1 and deallocates only the message block descriptor. If the `db_ref` count of the data block is 1, `freeb()` deallocates the message block descriptor and the data buffer.

If the message block was created via the `esballoc()` call, and if `db_ref` is 1, `freeb()` notifies the driver by invoking the `free_func` as pointed in the `free_rtn` structure. It waits for the driver to complete the `free_rtn` operation, and then proceeds to release the message block and data block.

RETURN VALUES

None

CONSTRAINTS

`freeb()` can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call.

NAME

freemsg () – Frees a STREAMS message.

SYNOPSIS

```
#include <sys/stream.h>
 void freemsg (mblk_t *mp);
```

PARAMETERS

mp	Pointer to the message to be freed.

DESCRIPTION

This function calls `freeb()` to free all message blocks and corresponding data blocks and data buffers associated with the message, beginning at *mp*.

RETURN VALUES

None

CONSTRAINTS

`freemsg()` can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call.

NAME

freezestr () – Freeze the state of a queue.

SYNOPSIS

```
#include <sys/stream.h>

 pl_t freezestr(queue_t *q);
```

PARAMETERS

q	Pointer to a message queue.

DESCRIPTION

For modules/drivers configured at a synchronization level other than `SQLVL_NOSYNC`, STREAMS/UX uses a different mechanism to protect the queues. The `freezestr()` utility in these modules/drivers returns the current interrupt priority level, and is only provided to make porting code from SVR4 MP easier.

For modules/drivers configured at the synchronization level `SQLVL_NOSYNC`, `freezestr()` freezes the state of the queue specified by q. Freezing a queue blocks everyone except the caller from adding or removing messages from the queue.

SQLVL_NOSYNC modules and drivers must freeze the queue before calling `insq()`, `rmvq()`, `strqget()`, and `strqset()`. Utilities like `getq()`, `putq()`, `putbq()`, `flushq()`, `qsize()` etc must not be called by the caller of `freezestr()`, while the queue is frozen, as they indirectly freeze the queue to ensure atomicity of queue manipulation. Calling `freezestr()` to freeze a queue that is already frozen by the same caller will result in deadlock.

Freezing the queue does not automatically stop all the functions that are already running in the stream. Instead, these functions will continue to run until they attempt to perform an operation that changes the state of the frozen queue. Then, they will be forced to wait for the queue to be unfrozen by a call to `unfreezestr()`.

It is the responsibility of the caller freezing the queue via `freezestr()`, to also unfreeze the queue by calling `unfreezestr()` after the required operations have been performed on the frozen queue.

`freezestr()` should be used sparingly. It is rarely necessary to freeze a queue as module/drivers do not need to manipulate the queues directly. Freezing a queue could have a significant negative impact on performance.

RETURN VALUES

The current interrupt priority level.

CONSTRAINTS

`freezestr()` can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call.

NAME

getadmin () – Returns a pointer to the module administration function.

SYNOPSIS

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

 int (*getadmin(ushort_t mid))();
```

PARAMETERS

mid	Module ID of module/driver, for which the administration function pointer is needed.

DESCRIPTION

`getadmin()` returns a pointer to the module administration function. The module administration function is pointed to by `qqadmin`, a member of the `qinit` structure. The module identifier is available in the `mi_idnum` field of the `module_info` structure. The `qi_minfo` field of the `qinit` structure points to the `module_info` structure. Note that while the `module_info` information may be available in both the read- and write-side `qinit` structures, but `getadmin()` returns the `admin()` function pointer from read-side `qinit` structure.

RETURN VALUES

`getadmin` returns a pointer to the module administration function if a module or driver is present with the specified identifier (*mid*). It returns NULL if no module or driver is present with that identifier.

CONSTRAINTS

`getadmin()` can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call.

NAME

getmid() – getmid returns the module identifier for a specified module or driver name.

SYNOPSIS

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

 ushort_t getmid (char *name);
```

PARAMETERS

name	Name of module/driver.

DESCRIPTION

This function looks into the `module_info` structure for the module/driver as specified in name and retrieves the `mi_idnum` value. Note that while the `module_info` information may be available in both the read-side and write-side `qinit` structures, but `getadmin()` looks only at the read-side `qinit`.

RETURN VALUES

`getmid()` returns the module identifier if a module/driver is found with the specified name, and a 0 if no matching module name is found.

CONSTRAINTS

`getmid()` can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call.

NAME

getq () – gets the next message from the head of the queue.

SYNOPSIS

```
#include <sys/stream.h>

 mblk_t *getq(queue_t *q);
```

PARAMETERS

q	Pointer to the queue from which the message is to be retrieved.

DESCRIPTION

`getq()` is used typically by the service procedure of a module/driver to retrieve the messages from the message queue.

RETURN VALUES

If there is a message left in the queue to be retrieved, `getq` returns a pointer to that message. If there are no messages in the queue, then `getq()` returns NULL and the queue is marked with `QWANTR` so that next time when a message is placed on the queue, it will be scheduled for service. If the data in the enqueued messages in the queue drops below the low water mark then queue behind the current queue is scheduled for service, if the `backq` previously attempted to place a message and failed.

CONSTRAINTS

`getq()` can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call.

NAME

insq () – To insert a message into a queue.

SYNOPSIS

```
#include <sys/stream.h>

 int insq (queue_t *q, mblk_t *mp, mblk_t *newmp);
```

PARAMETERS

q	Queue pointer.
mp	Pointer to message before which the new message is to be inserted.
newmp	Pointer to the new message to be inserted.

DESCRIPTION

The new message is inserted before the message pointed to by *mp*. The new message will be placed in an appropriate priority order, i.e., `insq()` will make an attempt to place the new message before, and as close as possible to the message pointed to by `mp`. If the new message priority does not allow the message to be placed before *mp*, `insq()` will fail. Further, the *mp* pointer needs to be a valid message pointer in the queue, otherwise the results will be unpredictable.

A queue belonging to a module/driver with a synchronization level of `SQLVL_NOSYNC` may be simultaneously manipulated by multiple threads. Therefore, the queue specified by q must be frozen by calling freezestr() before calling `insq()`, `rmvq()`, `strqget()`, and `strqset()`. A call to `unfreezestr()` must be made to unfreeze the queue after the above operations are complete.

RETURN VALUES

`insq()` returns a 1 on success and a 0 on failure.

CONSTRAINTS

`insq()` can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call.

NAME

`linkb`() – Concatenates two messages.

SYNOPSIS

```
#include <sys/stream.h>
 void linkb (mblk_t *mp1, mblk_t *mp2);
```

PARAMETERS

mp1	Pointer to first message block.
mp2	Pointer to the second message block that will be linked to *mp1*.

DESCRIPTION

`linkb()` links message *mp2* to message *mp1*. Data blocks and data buffers remain untouched. The `b_cont` pointer from the last message block of the message of which `mp1` is the first message block is updated to point to message block *mp2*.

RETURN VALUES

None

CONSTRAINTS

`linkb()` can be called from thread or interrupt context. Spinlocks can be held across this call.

NAME

LOCK () – Acquire a spinlock.

SYNOPSIS

```
#include <sys/stream.h>
 spl_t LOCK (lock_t *lockptr, spl_t prilev);
```

PARAMETERS

lockptr	Pointer to lock to be acquired.
prilev	Priority level to be set for the calling interrupt, while it holds the lock. In STREAMS/UX this parameter is ignored.

DESCRIPTION

`LOCK` acquires the lock specified by *lockp*. If the lock is not available the caller thread will busy wait until the lock becomes available.

RETURN VALUES

Returns the current `spl` level.

CONSTRAINTS

`LOCK()` can be called from thread or interrupt context. Spinlocks can be held across this call, provided lock orders are used correctly.

NAME

LOCK_ALLOC () – Allocates spinlock structure.

SYNOPSIS

```
#include <sys/stream.h> #include <sys/semglobal.h>

 lock_t *LOCK_ALLOC (uchar_t hierarchy, spl_t min_prilev,  lkinfo_t *lkinfoptr, int flag);
```

PARAMETERS

hierarchy	Hierarchy determines the order in which this lock is to be acquired. The STREAMS/UX `LOCK_ALLOC()` accepts the following hierarchy of parameter values. These are reserved for STREAMS/UX modules and drivers. • `STREAMS_USR1_LOCK_ORDER` • `STREAMS_USR2_LOCK_ORDER` • `STREAMS_USR3_LOCK_ORDER`
min_prilev	This value is ignored in STREAMS/UX.
lkinfoptr	Pointer to the `lkinfo` structure. `lk_name` in the `lkinfo` structure (`lkinfo_t`) contains the name of the lock.
flag	`KM_SLEEP` must be set.

DESCRIPTION

`LOCK_ALLOC()` is used to allocate and initialize a lock. It makes use of the native HP-UX `alloc_spinlock()` primitive.

RETURN VALUES

`LOCK_ALLOC()` returns a pointer to the allocated lock on success. The calling thread will block, if the memory is not available. If `KM_SLEEP` is not set, and memory is not available, `LOCK_ALLOC()` returns NULL.

CONSTRAINTS

`LOCK_ALLOC()` must be called from thread context. Spinlocks must not be held across the calls to `LOCK_ALLOC()`.

NAME

LOCK_DEALLOC () – Deallocates an acquired lock.

SYNOPSIS

```
#include <sys/stream.h>

 void LOCK_DEALLOC(lock_t *lockptr);
```

PARAMETERS

lockptr	Pointer to the lock that is to be deallocated.

RETURN VALUES

None

CONSTRAINTS

`LOCK_DEALLOC()` can be called from thread or interrupt context. Only spinlocks with STREAMS/UX user lock order can be held across this call.

NAME

msgdsize () – Gets the number of data bytes in a message.

SYNOPSIS

```
#include <sys/stream.h>
 int msgdsize (mblk_t *mp);
```

PARAMETERS

mp	Pointer to the message.

DESCRIPTION

`msgdsize()` returns the total number of bytes in a message block of type `M_DATA`.

RETURN VALUES

`msgdsize()` returns the number of bytes of data in the message.

CONSTRAINTS

`msgdsize()` can be called from thread or interrupt context. Spinlocks can be held across this call.

NAME

`msgpullup` () – Concatenates a specified number of bytes from a message into a new message.

SYNOPSIS

```
#include <sys/stream.h>

 mblk_t *msgpullup (mblk_t *mp, int len);
```

PARAMETERS

mp	Pointer to the message.
len	Number of bytes to be concatenated. If *len* is set to -1, all data bytes are concatenated.

DESCRIPTION

`msgpullup` concatenates and aligns the first *len* data bytes of the message pointed to by *mp*, copying the data into a new message. All message blocks that remain in the original message once *len* bytes have been concatenated and aligned (including any partial message blocks) are copied and linked to the end of the new message, so that the length of the new message is equal to the length of the original message. The original message is unaltered.

If *len* equals -1, all data are concatenated. If *len* bytes of the same message type cannot be found, `msgpullup()` fails and returns NULL.

RETURN VALUES

Upon success, `msgpullup()` returns a pointer to the new message. If the number of data bytes in the original message is less than *len* bytes or if memory allocation failures occur, `msgpullup` fails and returns NULL.

CONSTRAINTS

`msgpullup` can be called from thread or interrupt context. Only spinlocks with STREAMS/UX user lock order can be held across this call.

NAME

`noenable` () – Prevents a queue from being scheduled.

SYNOPSIS

```
#include <sys/stream.h>

 void noenable (queue_t *q);
```

PARAMETERS

q	Pointer to a queue.

DESCRIPTION

This function prevents the scheduling of the service procedure for the queue pointed by *q*. The `noenable()` does not prevent the queue's service procedure from being scheduled when a high priority message is enqueued, or by an explicit call to `qenable`.

`enableok()` can be used to schedule the service procedure, if it was previously disabled via `noenable()`.

RETURN VALUES

None

CONSTRAINTS

`noenable()` can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call.

NAME

OTHERQ () – Returns the pointer to queue’s partner queue.

SYNOPSIS

```
#include <sys/stream.h>

 queue_t *OTHERQ (queue_t *q);
```

ARGUMENTS

q	Pointer to the queue.

DESCRIPTION

The `OTHERQ()` returns a pointer to the other of the queue-pair structures that make up a STREAMS module/driver. If `q` points to the read queue, the write queue will be returned and vice versa.

RETURN VALUES

`OTHERQ()` returns a pointer to a queue’s partner.

CONSTRAINTS

`OTHERQ()` can be called from thread or interrupt context. Spinlocks can be held across this call.

NAME

pcmsg () – Checks if a specified message type is a priority control message.

SYNOPSIS

```
#include <sys/stream.h>

 int pcmsg (uchar_t type);
```

PARAMETERS

type	The type of the message to be tested.

DESCRIPTION

`pcmsg()` is used by put and service procedures of a module or driver. Typical use of `pcmsg` is for a put procedure to determine whether to process the message immediately or to place it on the queue for deferred processing.

The message type of a message is available in the `db_type` field of the `datab` structure.

RETURN VALUES

`pcmsg` returns a 1 if the type is high priority (`db_type >= QPCTL`). All other message types cause it to return a 0.

CONSTRAINTS

`pcmsg()` can be called from thread or interrupt context. Spinlocks can be held across this call.

NAME

pullupmsg () – pullupmsg concatenates multiple message blocks into a single message block.

SYNOPSIS

```
#include <sys/stream.h>

 int pullupmsg (mblk_t *mp, int len);
```

PARAMETERS

mp	Pointer to message whose message blocks are to be concatenated.
len	Number of bytes to be concatenated. If *len* = -1, all message blocks from the original message are concatenated into a single message block.

DESCRIPTION

pullupmsg() concatenates and aligns the number of bytes as represented by *len* bytes stored in a complex message *mp*. The pullupmsg only concatenates of same message type. If *len* is -1 then pullupmsg() will concatenate and align the entire contents of all the messages into a single data block.

RETURN VALUES

pullupmsg returns 1 on success, and a 0 on failure.

CONSTRAINTS

pullupmsg() can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call.

NAME

put () – Calls a driver or module’s put procedure.

SYNOPSIS

```
#include <sys/stream.h>

 void put(queue_t *q, mblk_t *mp);
```

PARAMETERS

q	Pointer to a queue.
mp	Pointer to a message block.

DESCRIPTION

This function is called by modules and drivers to invoke their own put procedure (for the specified queue) to operate on the specified message.

RETURN VALUES

None

CONSTRAINTS

`put()` can be called from thread or interrupt context. Spinlocks must not be held across calls to this function.

NAME

putbq() – Places a message back at the head of a queue.

SYNOPSIS

```
#include <sys/stream.h>

 int putbq (queue_t *q, mblk_t *mp);
```

PARAMETERS

q	Pointer to the queue.
mp	Pointer to the message.

DESCRIPTION

This function places the specified message at its assigned priority as close to the head of the queue as possible. It is normally called by the service procedure for the queue, and is invoked when a `canput()` or `bcanput()` call from the service procedure detects a flow control condition. All flow control parameters for the queue are updated. Note that high priority messages should not be placed back on a queue. They should be processed in spite of flow control.

RETURN VALUES

`putbq()` returns a 1 on success and a 0 on failure.

CONSTRAINTS

`putbq()` can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call.

NAME

putctl () – Send a control message to the queue.

SYNOPSIS

```
#include <sys/stream.h>

 int putctl(queue_t *q, int type);
```

PARAMETERS

q	The queue to which the message is sent.
type	The type of control message to be allocated and passed to the put procedure of the queue.

DESCRIPTION

This function allocates a message block and assigns the specified control message type. Then it invokes the put procedure for the specified queue with this newly allocated message. The putctl() returns an error if the specified message type is M_DATA, M_PROTO or M_PCPROTO, or if memory allocation fails.

RETURN VALUES

putctl returns 1 on success and 0 on failure.

CONSTRAINTS

putctl() can be called from thread or interrupt context. Spinlocks must not be held across the calls to putctl() function.

NAME

putctl1 () – Send a control message with one data byte.

SYNOPSIS

```
#include <sys/stream.h>

 int putctl1 (queue_t *q, int type, int parm);
```

PARAMETERS

q	Pointer to queue.
type	The type of the control message to be passed to the queue.

DESCRIPTION

`putctl1()` works exactly like `putctl()` with the addition that it allows a 1-byte parameter to be part of the control message. The parameter addition allows for stronger control message functionality.

RETURN VALUES

`putctl` returns 1 on success and 0 on failure.

CONSTRAINTS

`putctl1()` can be called from thread or interrupt context. Spinlocks must not be held across the calls to this function.

NAME

putctl2 () – Send a control message with two byte data.

SYNOPSIS

```
#include <sys/stream.h>

 int putctl2 (queue_t *q, int type, int parm1, int parm2);
```

PARAMETERS

q	Pointer to the queue to which the control message is to be sent.
type	Message type
parm1 & *parm2*	Two 1-byte data parameters

DESCRIPTION

putctl2() is an enhanced version of putctl1(). It allows for two 1-byte parameters to accompany the control message. Processing and error checking remains the same as putctl() or putctl1().

RETURN VALUES

putctl2 returns 1 on success and zero on failure.

CONSTRAINTS

putctl2() can be called from thread or interrupt context. Spinlocks must not be held across this call.

NAME

putnext () – To pass a message to the next queue.

SYNOPSIS

```
#include <sys/stream.h>

 int putnext (queue_t *q, mblk_t *mp);
```

PARAMETERS

q	Pointer to the queue from which this message is to be sent.
mp	Pointer to the message block to be sent.

DESCRIPTION

`putnext()` is used for passing messages to the next queue in a stream. This utility calls put procedure associated with the next queue (`q->q_next`) in a stream and passes it a message block pointer as an argument.

RETURN VALUES

None

CONSTRAINTS

`putnext()` can be called from thread or interrupt context. Spinlocks must not be held across this call.

NAME

putnextctl () – Send control message with one byte parameter to the next queue.

SYNOPSIS

```
#include <sys/stream.h>

 int putnextctl1 (queue_t *q, int type, int parm);
```

PARAMETERS

q	Pointer to the queue from which the control message is to be sent.
type	Type of the control message.
parm	1-byte data parameter.

DESCRIPTION

`putnextctl1()` behaves exactly like `putnextctl()` with the addition of a 1-byte parameter to the control message.

RETURN VALUES

On successful it returns 1. `putnext1()` returns 0 if, either the type specified is `M_DATA`, `M_PROTO`, or `M_PCPROTO`, or a message block cannot be allocated.

CONSTRAINTS

`putnextctl1()` does not block and can be called from thread or interrupt context. Spinlocks must not be held across this call.

NAME

putnextctl2 () – Send control message with two byte parameter to the next queue.

SYNOPSIS

```
#include <sys/stream.h>

 int putnextctl2 (queue_t *q, int type, int parm1, int parm2);
```

PARAMETERS

q	Pointer to the queue from which the control message is being sent.
type	Message type.
parm1 & *parm2*	Two one byte parameters.

DESCRIPTION

`putnextctl2()` behaves exactly like `putnextctl1()`, except that it allows for two 1-byte parameters in the control message instead of one.

RETURN VALUES

`putnextctl2()` fails and returns 0 if, either the type specified is a `M_DATA`, `M_PROTO`, or `M_PCPROTO`,or a message block cannot be allocated. It returns 1 if successful.

CONSTRAINTS

`putnextctl2()` does not block and can be called from thread or interrupt context. Spinlocks must not be held across this call.

NAME

putq () – Enqueue message on the queue for deferred processing.

SYNOPSIS

```
#include <sys/stream.h>

 int putq (queue_t *q, mblk_t *mp);
```

PARAMETERS

q	Pointer to the queue in which the message is to be put.
mp	Pointer to the message.

DESCRIPTION

`putq()` is typically used by put procedure for deferred processing of the messages. The `putq()` enqueues the message *mp* in the message queue based on its priority. The service procedure is enabled if a high priority message is enqueued and `QNOENAB` is unset. Flow control parameters are updated for this queue.

RETURN VALUES

`putq()` returns 1 on success and 0 on failure.

CONSTRAINTS

`putq()` can be called from thread or interrupt context. Spinlocks of STREAMS/UX user lock order can be held across this call, and should not pass driver's read queue or lower mux's write queue.

NAME

qenable () – Schedule service procedure to run.

SYNOPSIS

```
#include <sys/stream.h>

 void qenable (queue_t *q);
```

PARAMETERS

q	Pointer to the queue whose service procedure is to be scheduled.

DESCRIPTION

`qenable()` places the specified queue in a linked list of queues whose corresponding service procedure is to be executed by the STEAMS scheduler. When invoked, this function ignores the `noenable()` status of the queue, and schedules the service procedure.

RETURN VALUES

None

CONSTRAINTS

`qenable()` can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call.

NAME

`qprocsoff` () – Disable put and service procedures.

SYNOPSIS

```
#include <sys/stream.h>

void qprocsoff(queue_t *q);
```

PARAMETERS

q	Pointer to a read queue.

DESCRIPTION

STREAMS/UX only provides stubs which are no-ops for `qprocson` and `qprocsoff` to make porting easier.

RETURN VALUES

None

CONSTRAINTS

None

NAME

qprocson () – Enable put and service procedures.

SYNOPSIS

```
#include <sys/stream.h>

void qprocson(queue_t *q);
```

PARAMETERS

q	Pointer to a read queue.

DESCRIPTION

STREAMS/UX only provides stubs which are no-ops for `qprocson` and `qprocsoff` to make porting easier.

RETURN VALUES

None

CONSTRAINTS

None

NAME

qreply() – Sends a message back in a opposite direction.

SYNOPSIS

```
#include <sys/stream.h>

 void qreply (queue_t *q, mblk_t *mp);
```

PARAMETERS

q	Pointer to queue from which a message is to be passed in the opposite direction.
mp	Pointer to message to be passed as a reply.

DESCRIPTION

This function passes a message back in the opposite direction from the currently processing flow. A common use for `qreply()` is seen in the way positive or negative acknowledgments are sent in response to `M_IOCTL` messages received by a module or driver.

RETURN VALUES

None

CONSTRAINTS

`qreply()` can be called from thread or interrupt context. Spinlocks must not be held across calls to `qreply()`.

NAME

qsize () – Returns the number of messages in a queue.

SYNOPSIS

```
#include <sys/stream.h>

 int qsize (queue_t *q);
```

PARAMETERS

q	Pointer to queue.

RETURN VALUES

This function returns the number of messages in the specified queue. If the message queue is empty, 0 is returned.

CONSTRAINTS

`qsize()` can be called from thread or interrupt context. Spinlocks of STREAMS/UX user lock order can be held across calls to this function for modules/drivers configured with a synchronization level of `SQLVL_NOSYNC`. For modules/drivers configured with any other synchronization level, spinlocks that are not necessarily of the STREAMS/UX user lock order, can be held across calls to this function.

NAME

RD() – get a pointer to the read queue.

SYNOPSIS

```
#include <sys/stream.h>

 queue_t *RD(queue_t *q);
```

PARAMETERS

q	Pointer to the queue whose read queue is to be returned.

DESCRIPTION

The `RD()` macro accepts a queue pointer, *q*, as argument and returns a pointer to the read queue of the same module.

RETURN VALUES

The pointer to the read queue.

CONSTRAINTS

`RD()` can be called from thread or interrupt context. Spinlocks can be held across calls to this function.

NAME

rmvb () – Removes a message block from a message.

SYNOPSIS

```
#include <sys/stream.h>

 mblk_t *rmvb (mblk_t *mp, mblk_t *bp);
```

PARAMETERS

mp	Pointer to message from which message block is to be removed.
bp	Pointer to message block targeted for removal.

DESCRIPTION

`rmvb()` removes the message block pointed to by *bp* from the message pointed to by *mp*, and returns a pointer to the altered message. The message block is just removed from the message. It is the responsibility of the module or driver to actually free the message block and other related structures, as appropriate.

RETURN VALUES

If the block removed was the last (or only) block on the message, `rmvb()` returns a NULL, otherwise it returns a pointer to the altered message. If *bp* is not a valid block pointer for this message, `rmvb()` returns a -1.

CONSTRAINTS

`rmvb()` can be called from thread or interrupt context. Spinlocks can be held across calls to this function.

NAME

rmvq () – Removes a message from a queue.

SYNOPSIS

```
#include <sys/stream.h>

 void *rmvq (queue_t *q, mblk_t *mp);
```

PARAMETERS

q	Pointer to the queue from which a message is to be removed.
mp	Pointer to the message targeted for removal.

DESCRIPTION

`rmvq()` removes a message pointed to by *mp* from the queue specified by *q*. All flow control parameters for the queue are appropriately updated. If *mp* is not a valid message pointer for this queue, the results can unpredictable.

A queue belonging to a module/driver with a synchronization level of `SQLVL_NOSYNC` may be simultaneously manipulated by multiple threads. Therefore, the queue specified by *q* must be frozen by calling `freezestr()` before calling `insq()`, `rmvq()`, `strqget()`, and `strqset()`. A call to `unfreezestr()` must be made to unfreeze the queue after the above operations are complete.

RETURN VALUES

None

CONSTRAINTS

`rmvq()` can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call.

NAME

SAMESTR () – Test if next queue is of the same type.

SYNOPSIS

```
#include <sys/stream.h>
 int SAMESTR(queue_t *q);
```

PARAMETERS

q	Pointer to the queue.

DESCRIPTION

The `SAMESTR()` macro checks whether the next queue in a stream (if it exists) is of the same type (read/write) as the current queue. This macro can be used to determine the midpoint in a STREAMS-based pipe or welded streams where the read queue is linked to the write queue.

RETURN VALUES

`SAMESTR()` returns:

1 — if the next queue is of the same type as the current queue

0 — if the next queue does not exist or if it is not of the same type

CONSTRAINTS

`SAMESTR()` can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call.

NAME

streams_delay () – Delay process execution for a specified number of clock ticks.

SYNOPSIS

```
#include <sys/stream.h>
#include <sys/strenv.h>

 void streams_delay(int ticks);
```

PARAMETERS

ticks	The number of clock ticks to delay.

DESCRIPTION

`streams_delay()` causes the caller to sleep for at least the amount of time specified by *ticks*, which is in units of clock ticks. The exact length of the delay is not guaranteed but can be an arbitrary amount longer than requested due to the scheduling of other activity in the system.

RETURN VALUES

None

CONSTRAINTS

`streams_delay()` will block and so can be called from a thread context only. Spinlocks must not be held across this function call.

NAME

`streams_get_sleep_lock`() – Obtain the sleep lock.

SYNOPSIS

```
#include <sys/stream.h>
#include <sys/strenv.h>

 lock_t *streams_get_sleep_lock(caddr_t event);
```

PARAMETERS

event	Kernel address signifying an event for which the caller wishes to wait in sleep.

DESCRIPTION

`streams_get_sleep_lock()` should be called by modules and drivers in their open/close routines before sleeping to prevent missing wakeups. After calling this function, the open or close can release spinlocks before sleeping. Other processes cannot wakeup the open or close between the time it calls `streams_get_sleep_lock()` and sleeps through `streams_mpsleep()`.

Modules and drivers must not call the native HP-UX `get_sleep_lock()` primitive directly, because STREAMS/UX needs to do some additional synchronization before invoking `get_sleep_lock()`. Therefore, if they use `get_sleep_lock()`, modules and drivers must include `<sys/strenv.h>` (which redefines `get_sleep_lock` to `streams_get_sleep_lock`), to use `get_sleep_lock()`.

RETURN VALUES

`streams_get_sleep_lock` obtains a sleep spinlock, and returns a pointer to this lock.

CONSTRAINTS

`streams_get_sleep_lock()` can be called with a thread context only (that is from open/close routines). Spinlocks can be held across this function call but should be released before the thread can sleep.

NAME

streams_mpsleep () – Suspend process execution pending occurrence of an event.

SYNOPSIS

```
#include <sys/stream.h>
#include <sys/strenv.h>
```

```
int streams_mpsleep(caddr_t event, int pri, caddr_t mesg, int tmo, void *lockp, int flags);
```

PARAMETERS

event	Kernel address signifying an event for which the caller wishes to wait.
pri	Priority value the caller wishes to sleep at.
mesg	Not used, provided for portability.
tmo	Number of ticks after which the process executing needs to be resumed.
lockp	A pointer to the lock if held by the driver/module using `spinlock()` API's during this call.
flags	Not used, provided for portability.

DESCRIPTION

`streams_mpsleep()` must be used by the drivers or modules in their open and close routines to sleep instead of the native HP-UX sleep primitive. This function suspends the execution of a process to await the occurrence of an event. The process that called `streams_mpsleep()` can be resumed by a call to `wakeup()` with the same event specified as that used to call `streams_mpsleep()`.

The address event, has no significance except that the same address must be passed to:

— `streams_get_sleep_lock()` to acquire the sleep lock before sleeping. (`streams_mpsleep()` will verify and acquire the sleep lock if not already acquired by the driver or module when a call to this function is made.)

— `wakeup()` to resume the sleeping process

The value of the priority argument determines whether the sleep may be interrupted by signals. If the value of priority is less than or equal to the value of the constant `PZERO` (defined in `<sys/param.h>`), the sleeping process will not be awakened by a signal. If the value of priority is greater than `PZERO`, the process will wake up prematurely (without a call to wakeup) upon receipt of a non-ignored, non-held signal and will normally return 1 to the calling code.

Unlike the native HP-UX sleep, the caller can provide the duration of time in ticks (`tmo`) for which the process execution needs to be suspended through this function.

If the lock specified by `lockp` is held by the caller on entry, it will be released in `streams_mpsleep()`, before suspending the execution of the process. When the `streams_mpsleep()` returns to the caller, this lock will not be held.

RETURN VALUES

`streams_mpsleep()` returns:

0 — if the caller woke up because of a call to `wakeup ()`

1 — if a priority value greater than `PZERO` is specified and the process was interrupted and woken up prematurely.

`ETIME` — the process was woken up after *`tmo`* ticks elapsed.

CONSTRAINTS

`streams_mpsleep()` will block and so must only be called from the driver/module's open or close routine. Only spinlock specified by *`lockp`* and the sleep lock can be held on entry (these locks will be released by `streams_mpseleep()`). Spinlocks (with the exception of those previously mentioned) cannot be held across this call.

NAME

`streams_put` () – Allows non-STREAMS code to safely call STREAMS/UX utilities.

SYNOPSIS

```
#include <sys/stream.h>

 void streams_put(streams_put_t func, queue_t *q, mblk_t *mp, void *arg);
```

PARAMETERS

func	Function to be executed in the STREAMS context.
q	Queue pointer.
mp	Pointer to valid message block.
arg	Argument pointer to be passed to *func*.

DESCRIPTION

Non-STREAMS/UX code can call `streams_put`, passing it a function and a queue. STREAMS/UX runs the function as if it were the queue's put procedure. The function can safely manipulate the queue and access the same data structures as the queue's put procedure. STREAMS/UX passes `arg` to the function. The caller may pass any value in the argument. It is the responsibility of the caller to make sure that the queue being specified is valid.

It is typically called by driver's interrupt service routine or timeout or `bufcall` callback function.

RETURN VALUES

None

CONSTRAINTS

`streams_put()` can be called from thread or interrupt context. Spinlocks must not be held across this function call.

NAME

`streams_time` () – Get time.

SYNOPSIS

```
#include <sys/stream.h> #include <sys/strenv.h>
 time_t streams_time();
```

PARAMETERS

None

DESCRIPTION

`streams_time()` returns the value of time in seconds since the Epoch.

RETURN VALUES

`streams_time()` returns the value of time in seconds since the Epoch.

CONSTRAINTS

`streams_time()` can be called from thread or interrupt context. Spinlocks must not be held across this function call.

NAME

streams_timeout () – Schedule a timeout to execute a specified function after a specified time delay.

SYNOPSIS

```
#include <sys/stream.h>
#include <sys/strenv.h>
```

```
toid_t *streams_timeout (timeout_fcn_t func, timeout_arg_t arg, int ticks, pl_t prilev, int
call_mp_timeout);
```

PARAMETERS

func	Function to be executed after a specified time interval.
arg	Argument passed to the timeout handler.
ticks	Number of ticks after which the function is to be executed.
prilev	Interrupt priority level at which the function will be called. The valid priority levels are `pltimeout`, `plstr`, `plhi` and `invpl`.
call_mp_timeout	Set 1 to indicate that the caller is MP aware STREAMS module/driver.

DESCRIPTION

The `streams_timeout()` function executes the specified function `func()` after the time interval as specified in ticks have expired. It returns an integer identification number. The `streams_untimeout()` function cancels a timeout request using this identification number.

RETURN VALUES

`streams_timeout()` returns a non-zero identifier to be passed to `streams_untimeout()` function to cancel the request, if the function was successfully scheduled for execution. If invalid interrupt priority levels are passed, then `streams_timeout()` returns a zero.

CONSTRAINTS

`streams_timeout()` can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call.

NAME

streams_untimeout () – Cancel pending timeout request scheduled through `streams_timeout()`.

SYNOPSIS

```
#include <sys/stream.h>
#include <sys/strenv.h>

 int streams_untimeout(toid_t id);
```

PARAMETERS

id	Non-zero identifier returned from a prior call to `streams_timeout()`.

DESCRIPTION

The `streams_untimeout()` cancels the pending timeout request specified by *id*, scheduled through a call to `streams_timeout()` earlier.

RETURN VALUES

This function returns a non-zero value indicating the time (in ticks) remaining on the timer on successfully cancelling the request and returns a -1 if the timeout request is not found.

CONSTRAINTS

`streams_untimeout()` can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call.

NAME

strlog () – Submit messages for logging to streams log driver.

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stream.h>
#include <sys/strlog.h>

 int strlog(short mid, short sid, char level, ushort_t flags, char *fmt, ... /* args */);
```

PARAMETERS

mid	STREAMS module id for the module/driver submitting the message for logging.
sid	Refers to the sub-ID number of a minor device of the driver associated with the STREAMS module or driver identified by mid.
level	Specifies a level for selective screening of lower-level event messages from a tracer.
flags	Contains several flags that can be set in various combinations. The flags are as follows:

`SL_FATAL`	Provides a notification of a fatal error.
`SL_NOTIFY`	Makes a request to mail a copy of a message to the system administrator.
`SL_ERROR`	The message is for the error logger.
`SL_TRACE`	The message is for the tracer.
`SL_CONSOLE`	The message will be printed to the console.

The following are additional flags. The `strlog` interface does not use these flags:

`SL_WARN`	The message is a warning.
`SL_NOTE`	The message is a note.

fmt	`printf` style format string. This accepts the `%x`, `%l`, `%o`, `%u`, `%d`, `%c`, and `%s` conversion specifications.
args	Are numeric or character arguments for the format string. There is no maximum number of arguments that can be specified.

DESCRIPTION

`strlog()` submits messages containing specified information to the streams log driver, *strlog* (7). The messages can be retrieved with the `getmsg()` system call. The flags argument specifies the type of the message and where it is to be sent. *strace* (1M) receives messages from the log driver and sends them to the standard output. *strerr* (1M) receives error messages from the log driver and appends them to a file called `/var/adm/streams/error.mm-dd`, where `mm-dd` identifies the date of the error message.

RETURN VALUES

None

CONSTRAINTS

`strlog()` can be called from thread or interrupt context. Spinlocks must not be held across this call.

NAME

strqget () – Retrieves information about a queue or priority band of the queue.

SYNOPSIS

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

int strqget (queue_t *q, qfields_t what, uchar_t band, long *valp);
```

PARAMETERS

q	Pointer to the queue.
what	The field of the queue about which to return information. Valid what values are:

QHIWAT	High water mark of the specified priority band.
QLOWAT	Low water mark of the specified priority band.
QMAXPSZ	Maximum packet size of the specified priority band.
QMINPSZ	Minimum packet size of the specified priority band.
QCOUNT	Number of bytes of data in messages in the specified priority band.
QFIRST	Pointer to the first message in the specified priority band.
QLAST	Pointer to the last message in the specified priority band.
QFLAG	Flags for the specified priority band.

band	Priority band of the queue about which to obtain information.
valp	Pointer to the memory location where the value is to be stored.

DESCRIPTION

This function provides modules and drivers a way to retrieve various queue and queue-band parameters without directly dereferencing the queue data structure.

A queue belonging to a module/driver with a synchronization level of `SQLVL_NOSYNC` may be simultaneously manipulated by multiple threads. Therefore, the queue specified by *q* must be frozen by calling `freezestr()` before calling `insq()`, `rmvq()`, `strqget()`, and `strqset()`. A call to `unfreezestr()` must be made to unfreeze the queue after the above operations are complete.

RETURN VALUES

`strqget()` returns 0 on success and the actual value of the requested field is stored in the memory pointed to by *valp*. Upon failure, it returns one of the following error codes.

EINVAL	If the band specified in `pri` does not exist.
ENOENT	If `QBAD` is specified or unidentified value is passed to `strqget()`.

CONSTRAINTS

`strqget()` can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call.

NAME

strqset () – Set the information in a queue or a queue band.

SYNOPSIS

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

int strqset (queue_t *q, qfields_t what, uchar_t pri, long newval);
```

PARAMETERS

q	Pointer to the queue.
what	The field of the queue about which to return information. Valid what values are:

QHIWAT	High water mark of the specified priority band.
QLOWAT	Low water mark of the specified priority band.
QMAXPSZ	Maximum packet size of the specified priority band.
QMINPSZ	Minimum packet size of the specified priority band.

pri	Priority band of a queue to be modified
newval	The new value to be set in queue fields.

DESCRIPTION

This function provides modules and drivers with a way to easily update the different (modifiable) queue and queue-band parameters without directly dereferencing queue pointers.

A queue belonging to a module/driver with a synchronization level of SQLVL_NOSYNC may be simultaneously manipulated by multiple threads. Therefore, the queue specified by *q* must be frozen by calling `freezestr()` before calling `insq()`, `rmvq()`, `strqget()`, and `strqset()`. A call to `unfreezestr()` must be made to unfreeze the queue after the above operations are complete.

RETURN VALUES

`strqset()` returns 0 on success. Upon failure, one of the following error codes is returned.

EINVAL	If the band specified in `pri` does not exist.
EPERM	If caller specified any one of the following fields QCOUNT QFIRST QLAST QFLAG
ENOENT	If QBAD is specified or unidentified value is passed to `strqset()`.

CONSTRAINTS

`strqset()` can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call.

NAME

SV_ALLOC () – Allocate and initialize a synchronization variable.

SYNOPSIS

```
#include <sys/stream.h>

 sv_t *SV_ALLOC(int flag);
```

PARAMETERS

flag Specifies whether the caller is willing to sleep waiting for memory. Valid flags are:

KM_SLEEP	Caller willing to sleep.
KM_NOSLEEP	Caller cannot sleep.

DESCRIPTION

SV_ALLOC dynamically allocates and initializes an instance of a synchronization variable. Synchronization variables are synchronization primitives that are used to coordinate the execution of processes based on asynchronous events. When allocated, these variables serve as points on which one or more processes can block until an event occurs. Then one or all the processes can be unblocked at the same time. Any function that blocks on a synchronization variable must be able to tolerate premature wakeups. If flag is set to KM_SLEEP, the caller will sleep if necessary until sufficient memory is available. If flag is set to KM_NOSLEEP, the caller will not sleep, but SV_ALLOC will return NULL if sufficient memory is not immediately available.

RETURN VALUES

Upon successful completion, SV_ALLOC returns a pointer to the newly allocated synchronization variable. If KM_NOSLEEP is specified and sufficient memory is not immediately available, SV_ALLOC returns a NULL pointer.

CONSTRAINTS

If KM_SLEEP is specified, SV_ALLOC() may sleep and can only be called from a thread context. If KM_NOSLEEP is specified, spinlocks of STREAMS/UX user lock order can be held across this call and may be called from thread or interrupt context.

NAME

SV_BROADCAST () – Wake up all processes sleeping on a synchronization variable.

SYNOPSIS

```
#include <sys/stream.h>
 void SV_BROADCAST(sv_t *svp, int flags);
```

PARAMETERS

svp	Pointer to the synchronization variable to be broadcast signaled.
flags	Bit field for flags. No flags are currently defined for use in drivers and the flags argument must be set to zero.

DESCRIPTION

If one or more processes are blocked on the synchronization variable specified by `svp`, `SV_BROADCAST` wakes up all of the blocked processes. Note that synchronization variables are stateless, and therefore calls to `SV_BROADCAST` only affect processes currently blocked on the synchronization variable and have no effect on processes that block on the synchronization variable at a later time.

RETURN VALUES

None

CONSTRAINTS

`SV_BROADCAST()` can be called from the thread or interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call.

NAME

SV_DEALLOC () – Deallocate an instance of a synchronization variable.

SYNOPSIS

```
#include <sys/stream.h>

 void SV_DEALLOC(sv_t *svp);
```

PARAMETERS

svp	Pointer to the synchronization variable to be deallocated.

DESCRIPTION

`SV_DEALLOC` deallocates the synchronization variable specified by *svp*.

RETURN VALUES

None

CONSTRAINTS

`SV_DEALLOC()` does not sleep and so can be called from thread and the interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call.

NAME

SV_WAIT () – Sleep on a synchronization variable.

SYNOPSIS

```
#include <sys/stream.h>

 void SV_WAIT(sv_t *svp, int priority, lock_t *lkp);
```

PARAMETERS

svp	Pointer to the synchronization variable on which to sleep.
priority	The priority value passed here is subtracted from PZERO-1. pridisk, prinet, pritty, pritape, prihi, primed and prilo are defined to be 0 and do not affect the caller's priority. To change the process's priority, study the priorities in /usr/include/sys/param.h and pass the needed offset to PZERO-1 in the priority parameter.
lkp	Pointer to a basic lock which must be locked when SV_WAIT is called. The basic lock is released when the calling process goes to sleep.

DESCRIPTION

SV_WAIT causes the calling process to sleep (the caller's execution is suspended and other processes may be scheduled) waiting for a call to *SV_BROADCAST* (D3) for the synchronization variable specified by *svp*.

The basic lock specified by lkp must be held by the caller upon entry. When SV_WAIT returns, the lkp spinlock is not held and SV_WAIT lowers the priority level to the value before the caller acquired the lkp spinlock, which may not be SPLNOPREEMPT. If the caller acquired the lock while holding other spinlocks, the priority level is lowered to the value before the first of these nested spinlock calls.

RETURN VALUES

None

CONSTRAINTS

SV_WAIT() sleeps and can only be called from the thread context. Spinlocks (with the exception of the lock specified in lkp) cannot be held across this call.

NAME

SV_WAIT_SIG() – Sleep on a synchronization variable.

SYNOPSIS

```
#include <sys/stream.h>

 bool_t SV_WAIT_SIG(sv_t *svp, int priority, lock_t *lkp);
```

PARAMETERS

svp	Pointer to the synchronization variable on which to sleep.
priority	The priority value passed here is added to `PZERO+1` and ORed with `PCATCH`. `pridisk`, `prinet`, `pritty`, `pritape`, `prihi`, `primed` and `prilo` are defined to be 0 and do not affect the caller's priority. To change the process's priority, study the priorities in `/usr/include/sys/param.h` and pass the needed offset to `PZERO+1` in the priority parameter.
lkp	Pointer to a basic lock which must be locked when `SV_WAIT_SIG` is called. The basic lock is released when the calling process goes to sleep.

DESCRIPTION

`SV_WAIT_SIG()` causes the calling process to sleep waiting for a call to *SV_BROADCAST* (D3) for the synchronization variable specified by `svp`. `SV_WAIT_SIG()` may be interrupted by a signal, in which case it returns early without waiting for a call to `SV_BROADCAST()`.

The basic lock specified by `lkp` must be held by the caller upon entry. When `SV_WAIT` returns, the `lkp` spinlock is not held and `SV_WAIT` lowers the priority level to the value before the caller acquired the `lkp` spinlock, which may not be `SPLNOPREEMPT`. If the caller acquired the lock while holding other spinlocks, the priority level is lowered to the value before the first of these nested spinlock calls.

RETURN VALUES

`SV_WAIT_SIG()` returns TRUE (a non-zero value) if the caller woke up because of a call to `SV_BROADCAST()`. `SV_WAIT_SIG()` returns FALSE (zero) if the caller woke up and returned early because of the sleep being interrupted.

CONSTRAINTS

`SV_WAIT_SIG()` sleeps and can only be called from the thread context. Spinlocks (with the exception of the lock specified in `lkp`) cannot be held across this call.

NAME

TRYLOCK () – Try to acquire a basic lock.

SYNOPSIS

```
#include <sys/stream.h>
 spl_t TRYLOCK(lock_t *lockp, spl_t pl);
```

PARAMETERS

lockp	Pointer to the basic lock to be acquired.
pl	This value is ignored, and retained for portability only.

DESCRIPTION

If the lock specified by `lockp` is immediately available (can be acquired without waiting) `TRYLOCK` acquires the lock. If the lock is not immediately available, the function returns without acquiring the lock.

RETURN VALUES

If the lock is acquired, `TRYLOCK` returns the previous interrupt priority level (`plbase - plhi`). If the lock is not acquired the value `invpl` is returned.

CONSTRAINTS

`TRYLOCK()` can be called from the thread or the interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across the calls.

NAME

`testb` () – Tests for an available buffer.

SYNOPSIS

```
#include <sys/stream.h>

 int testb (int bufsize, unsigned int pri);
```

PARAMETERS

bufsize	Message buffer size in bytes.
pri	This field is obsolete.

DESCRIPTION

`testb()` tests if a STREAMS message of specified size can be allocated. It returns 1 if memory is available, otherwise it returns 0. Successful return from this call does not guarantee successful memory allocation for a subsequent `allocb()` call.

CONSTRAINTS

`testb()` can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call.

NAME

`unbufcall` () – Cancels a pending `bufcall()` or `esbcall()` request specified by ID.

SYNOPSIS

```
#include <sys/stream.h>
 void unbufcall (toid_t id);
```

PARAMETERS

id	A Non zero identifier returned from a prior `bufcall()` or `esbbcall()`.

DESCRIPTION

If `unbufcall()` is invoked while any function called by the pending `bufcall()` or `esbbcall()` request is running, `unbufcall()` will not return until the function completes or the pending request is cancelled.

RETURN VALUES

None

CONSTRAINTS

`unbufcall()` can be called from thread or interrupt context. Spinlocks must not be held across this function call.

NAME

`unfreezestr` () – Unfreeze the frozen queue.

SYNOPSIS

```
#include <sys/stream.h>

 void unfreezestr(queue_t *q, pl_t pl);
```

PARAMETERS

q	Pointer to a message queue.
pl	The interrupt priority level to be set after unfreezing the stream. (This value is ignored, but has been retained for portability.)

DESCRIPTION

`unfreezestr()` unfreezes the queue specified by *q*. Unfreezing the queue allows the continuation of all activities that may have been forced to wait while the queue was frozen. This call must only be made by the caller of `freezestr()`.

The caller freezing the queue by calling `freezestr()` must also unfreeze the queue by calling unfreezestr() after the required operations have been performed on the frozen queue.

`unfreezestr()` is only applicable to the module/drivers configured at the synchronization level `SQLVL_NOSYNC`. For modules/drivers configured at any other synchronization level, STREAMS/UX uses a different mechanism to protect STREAMS/UX queues and the `unfreezestr()` utility for these modules/drivers is a no-op provided to make porting code easier.

RETURN VALUES

None

CONSTRAINTS

None

NAME

unlinkb() – Removes the first message block from the head of a message.

SYNOPSIS

```
#include <sys/stream.h>

 mblk_t *unlinkb(mblk_t *mp);
```

PARAMETERS

mp	Pointer to message from which the first block is to be removed.

DESCRIPTION

`unlinkb()` will unlink the first message block from the message and the altered message is returned to the caller. It is the responsibility of the module or driver to actually free the message block.

RETURN VALUES

`unlinkb()` returns a pointer to the head of the altered message after unlinking the first block. If this was the last (or only) block in the message, `unlinkb()` returns a NULL pointer.

CONSTRAINTS

`unlinkb()` can be called from thread or interrupt context. Spinlocks can be held across this call.

NAME

UNLOCK () – Release the previously acquired lock.

SYNOPSIS

```
#include <sys/stream.h>
 void UNLOCK (lock_t *lockptr, spl_t prilev);
```

PARAMETERS

lockptr	Pointer to lock targeted for release.
prilev	This parameter is ignored as STREAMS always acquires the spinlock at `SPL6`.

DESCRIPTION

`UNLOCK()` calls the native HP-UX `spinunlock` primitive to release the previously acquired lock.

RETURN VALUES

None

CONSTRAINTS

`UNLOCK()` can be called from thread or interrupt context. Spinlocks can be held across the calls to `UNLOCK()`.

NAME

unweldq () – Disconnects previously established weld connection.

SYNOPSIS

```
#include <sys/stream.h>

int unweldq    (queue_t *d1_wq, queue_t *d2_rq, queue_t *d2_wq, queue_t *d1_rq, weld_fcn_t
               func, weld_arg_t arg, queue_t *protect_q);
```

PARAMETERS

d1_wq, d1_rq	First drivers' write and read queues.
d2_wq, d2_rq	Second drivers' write and read queues
func and arg	Optional callback function pointer and argument for callback.
protect_q	Optional synchronization queue to be acquired when *func* is executed.

DESCRIPTION

unweldq() operation sets d1_wq->q_next and d2_wq->q_next to NULL there by disconnecting the previously established connection. unweldq() function does not actually perform this operation, instead it sends an asynchronous request to the weld daemon to perform unweld operation on the specified queues.If the *func* and protect_q are specified, the weld daemon acquires the synchronization queue before executing the *func*.

However, if the caller does not need to be notified by the weld daemon, the parameters *func*, *arg*, and *protect_q* can be set to 0.

RETURN VALUES

unweldq() returns 0 on success and one of the error codes upon failure.

ENXIO	Weld mechanism is not available.
EINVAL	Invalid queue arguments.
EAGAIN	Could not allocate weld record.

CONSTRAINTS

unweldq() can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call.

NAME

vtop () – Convert virtual address to physical address.

SYNOPSIS

```
#include <sys/stream.h>

 paddr_t vtop(caddr_t vaddr, proc_t *p);
```

PARAMETERS

vaddr	Virtual address to convert.
p	Pointer to the process structure used. To indicate that the address is in kernel virtual space, p must be set to NULL.

DESCRIPTION

`vtop()` converts a virtual address to a physical address.

RETURN VALUES

On success, `vtop()` returns the physical address. Otherwise, if no physical memory is mapped to the virtual address or the pointer to the `proc_t` is not NULL, `vtop()` returns 0.

CONSTRAINTS

`vtop()` can be called from thread or interrupt context. Spinlocks can be held across this call.

NAME

WR() – Get pointer to the write queue.

SYNOPSIS

```
#include <sys/stream.h>

 queue_t *WR(queue_t *q);
```

PARAMETERS

q	Pointer to the queue whose write queue is to be returned.

DESCRIPTION

The macro `WR()` accepts a queue pointer as an argument and returns a pointer to the write queue of the same module.

RETURN VALUES

The pointer to the write queue.

CONSTRAINTS

`WR()` can be called from thread or interrupt context. Spinlocks can be held across the calls to WR`()`.

NAME

weldq () – Establish connections between two drivers' queues.

SYNOPSIS

```
#include <sys/stream.h>

int weldq      (queue_t *d1_wq, queue_t *d2_rq, queue_t *d2_wq, queue_t *d1_rq, weld_fcn_t
               func, weld_arg_t arg, queue_t *protect_q);
```

PARAMETERS

d1_wq, *d1_rq*	First drivers' write and read queues.
d2_wq, *d2_rq*	Second drivers' write and read queues
func and *arg*	Optional callback function pointer and argument for callback.
protect_q	Optional synchronization queue to be acquired when `func` is executed.

DESCRIPTION

`weldq() sets d1_wq->q_next` to point to `d2_rq` and `d2_wq->q_next` to point to `d1_rq`. It sends an asynchronous request to the weld daemon to update relevant queue parameters and returns back to the caller without waiting for the completion of that request. If the *func* and `protect_q` are specified, the weld daemon acquires the synchronization queue before executing the `func`.

However, if the caller does not need to be notified by the weld daemon, the parameters *func*, *arg*, and *protect_q* can be set to 0.

RETURN VALUES

`unweldq()` returns 0 on success and one of the error codes upon failure.

`ENXIO`	Weld mechanism is not available.
`EINVAL`	Invalid queue arguments.
`EAGAIN`	Could not allocate weld record.

CONSTRAINTS

`weldq()` can be called from thread or interrupt context. Only spinlocks of STREAMS/UX user lock order can be held across this call.

C Message Types

This section describes the fixed set of message types recognized by STREAMS/UX. As described in *Chapter 3, "Messages,"* STREAMS messages can be classified as ordinary (non-priority) messages that are subject to flow control, and high priority messages that are not subject to flow-control.

STREAMS defined message types differ in their intended purposes, their treatment at the stream head, and their message queueing priority. STREAMS does not prevent a module or driver from generating any message type or sending it in any direction on the stream. However, established processing and direction rules should be observed. Stream head processing is fixed according to the message type, although certain parameters can be altered.

Ordinary Messages

M_BREAK

This message is sent to a driver to transmit a `BREAK` on the device controlled by the driver. The message format for an `M_BREAK` is defined by the driver developer. This message is never generated from a user process, and is discarded when received by the stream head. This message may be considered as a special case of the `M_CTL` message.

M_CTL

This message (and its corresponding priority counterpart `M_PCCTL`) is typically used by modules to send specific information to other modules. The use of this message type is driven by module functionality. This message is never generated from a user process, and is discarded when received by the stream head.

M_DATA

An `M_DATA` message contains normal data. It is the default message type for message blocks allocated via the `allocb()` function. For messages with multiple message blocks, the message type for all messages following the first `M_DATA` block will be `M_DATA`. In the *putmsg* (2) and *getmsg* (2) system calls, the contents of `M_DATA` message blocks are referred to as the data part. `M_DATA` can be sent bi-directionally on a Stream by STREAMS components as well as a user process.

M_DELAY

This message is sent to a driver to request a real-time delay on output, typically to avoid exceeding the buffer size of devices (for example, slow terminal devices). `M_DELAY` can be seen as a special case of the `M_CTL` message type, and its usage is developer-dependent. Not all devices may recognize this message. This message is never generated from a user process, and is discarded if received by the stream head.

M_IOCTL

This message is generated by the stream head in response to `I_STR`, `I_LINK`, `I_UNLINK`, `I_PLINK`, `I_PUNLINK`, and IOCTL calls that are not defined in *streamio* (7). When the stream head receives one of these IOCTL calls, it creates an `M_IOCTL` message by using the values supplied in the `ioctl()` system call and the process that issued the IOCTL system call. The `M_IOCTL` message is then sent downstream.

For an `I_STR` IOCTL call, the user process sets the *cmd* parameter to `I_STR`, and the *arg* parameter to a buffer in the user space of type `strioctl`. This is defined in `<stropts.h>` which contains following fields:

```
int     ic_cmd;             /* downstream command */

int     ic_timout;          /* ACK/NAK timeout */

int     ic_len;             /* length of data arg */

char *  ic_dp;              /* ptr to data arg */
```

Where ic_cmd describes the command intended for module or driver, ic_timout specifies the number of seconds that the I_STR request will wait for an acknowledgement before timing out. ic_dp points to the data buffer, and ic_len is the length of the data buffer passed in and, on return from the call, it contains the length of the data (if any) in the being returned to the user.

An M_IOCTL message consists of an M_IOCTL message block followed by zero or more M_DATA blocks. The M_IOCTL message block contains an iocblk structure defined in <sys/stream.h> and has following fields:

```
int             ioc_cmd;        /* ioctl command type           */
cred_t *        ioc_cr;         /* pointer to full credentials  */
uint            ioc_id;         /* ioctl id                     */
uint            ioc_flag;       /* see flag values below        */
ioc_pad         ioc_cnt;        /* count of bytes in data field */
int             ioc_error;      /* error code                   */
int             ioc_rval;       /* return value                 */
```

For an I_STR IOCTL, ioc_cmd corresponds to ic_cmd of the strioctl structure. The ioc_cr points to a credentials structure (see <sys/cred.h>) defining the process's permissions. The ioc_cnt is the number of data bytes, if any, contained in the message and corresponds to ic_len. The ioc_id is an internally generated identifier used by stream head to match each M_IOCTL message sent downstream, with responses that come upstream to the stream head. An M_IOCACK or M_IOCNACK response message completes the IOCTL processing.

For I_STR IOCTL, the user supplied data to be sent to the module or driver is attached as zero or more M_DATA messages and is linked to the initial M_IOCTL message block. The ioc_cnt is copied from ic_len. If there is no data, ioc_cnt is zero.

When the stream head does not recognize an IOCTL command, it creates a transparent M_IOCTL message. The M_IOCTL message for transparent processing consists of an M_IOCTL message block followed by one M_DATA message block containing the address of the third argument to the ioctl() system call. The form of the iocblk structure is same as previously noted. However, ioc_cmd is set to the user-specified IOCTL command and ioc_cnt is set to a special constant TRANSPARENT defined in <sys/stream.h>. TRANSPARENT recognizes the transparent nature of the M_IOCTL message when an I_STR IOCTL may specify a value of ioc_cmd equivalent to the user-specified IOCTL command argument of a transparent IOCTL. The first module to recognize the IOCTL command in ioc_cmd will process the M_IOCTL message. Intermediate modules that do not recognize the IOCTL command must pass the message on. If, the driver does not recognize the IOCTL command, it must respond with an M_IOCNAK message to stream head.

The M_IOCACK and M_IOCNAK message types have the same format as an M_IOCTL message and contain an iocblk structure in the first block. An M_IOCACK block may be linked to zero or more M_DATA blocks. If an M_IOCACK or M_IOCNAK message reaches the stream head with an identifier, which does not match that of the currently outstanding M_IOCTL message, the response message is discarded. Usually, to assure a correct response, the replying module converts the M_IOCTL message into an M_IOCACK or M_IOCNAK message and returns to stream head. If no data is returned, the ioc_cnt in the M_IOCACK message is set to zero. The ioc_error is set to any return error condition by the downstream module. If this value is non-zero, it is returned to the user in errno. A return value can be specified only with M_IOCACK message by setting ioc_rval field. For M_IOCNAK messages stream head ignores ioc_rval field.

For I_STR IOCTL, if the module wants to send data to the user process, it must use the M_IOCACK message to send the data by linking one or more M_DATA blocks to it and setting ioc_cnt to number of data bytes. The stream head places the data in the address pointed to by ic_dp in the user I_STR strioctl structure.

For transparent IOCTL, if the module wants to send data to the user process, it must use an M_COPYOUT message. The stream head will ignore any data contained in the M_IOCACK message.

No data can be sent to the user with an M_IOCNAK message. The stream head will free any M_DATA blocks linked to the M_IOCNAK message block.

If the stream head does not receive an M_IOCACK or M_IOCNAK message in response to an M_IOCTL (same ioc_id) message, it will block for all IOCTL calls except for I_STR IOCTL. For I_STR IOCTL if an M_IOCACK or M_IOCNAK message is not received as a response to an M_IOCTL (same ioc_id) message, it will fail when the timeout specified in ic_timout expires (and block when the timeout is infinite).

M_PASSFP

This message is used by STREAMS to pass a file pointer from the stream head at one end of a STREAMS-based pipe to the stream head at the other end of the same pipe.

This message is generated as a result of an I_SENDFD IOCTL issued by a process to the sending stream head. STREAMS places the M_PASSFP message directly on the read-queue of the other stream head. The user process retrieves the file pointer contained in M_PASSFP messages through the I_RECVFD ioctl() command. This message type is restricted to the stream head and should be only processed by the stream head read queue. This message can be ignored by module and driver developers.

M_PROTO

This message contains control information and associated data. The message format is one or more M_PROTO message blocks followed by one or more M_DATA message blocks. The semantics of the M_DATA and M_PROTO message blocks are determined by the STREAMS module that receives the message. M_PROTO messages can travel bi-directionally on a stream and can be passed between a process and the stream head. The contents of the first message block are generally referred as the control part, and the contents of any following M_DATA message blocks are referred as the data part.

Note that on the write-side, the user can generate M_PROTO messages containing only one M_PROTO message. Also, it is recommended that on the read-side, the format of M_PROTO and M_PCPROTO messages should generally contain only one M_PROTO or M_PCPROTO message block. The *getmsg* (2) will compact the multiple M_PROTO/M_PCPROTO message blocks into one single control part when delivering the message to the user process.

M_RSE

Reserved for internal use. Modules that do not recognize this message must pass it on. Drivers that do not recognize this message must free it.

M_SETOPTS

This message is used to alter the characteristics of the stream head. It is generated by any downstream module and is interpreted by the stream head. The data buffer in the first message block consists of a stroptions structure. This structure is defined in <sys/stropts.h> and shown here:

```
ulong so_flags;         /* options to set */
short so_readopt;       /* read option */
ushort so_wroff;        /* write offset */
long so_minpsz;         /* minimum read packet size */
long so_maxpsz;         /* maximum read packet size */
ulong so_hiwat;         /* read queue high-water mark */
ulong so_lowat;         /* read queue low-water mark */
unsigned char so_band;  /* update water marks for this band */
```

Where so_flags specifies the options to be altered. The options can be any combination of the following:

SO_ALL	Update all options according to the values specified in the remaining fields of the stroptions structure.
SO_READOPT	Set the read mode as specified in the so_readopt field. The read modes are:

RNORM:	Byte stream
RMSGD:	Message discard
RMSGN:	Message non-discard
RPROTDAT:	Convert M_PROTO/M_PCPROTO into M_DATA
RPROTNORM:	Normal protocol
RPROTDIS:	Discard M_PROTO/M_PCPROTO message blocks and retain any linked M_DATA blocks

SO_WROFF	Insert an offset (in bytes) specified in the so_wroff field of the stroptions structure into the first message block of all M_DATA messages created by the *write* (2) system call, and into the first M_DATA message blocks, if any, of all messages created by the *putmsg* (2) system call. Write-offsets must not exceed the maximum size of the message data buffer. The default offset is zero. Further, modules and drivers must verify, that the b_rptr in the **msgb** structure is greater than the db_base in the **datab** structure, to determine that an offset has been inserted in the first message block.
SO_MINPSZ	Set the minimum packet size for the stream head read-queue to the so_minpsz value in the stroptions structure. This value is advisory for modules immediately below the stream head. This is intended to limit the size of M_DATA messages that the module should put to the stream head. There is no intended minimum size for other message types. The default value in the stream head is zero.
SO_MAXPSZ	Set the maximum packet size for the stream head read-queue to the so_maxpsz value in the stroptions structure. This value is advisory for modules immediately below the stream head. This is intended to limit the size of M_DATA messages that the module should put to the stream head. There is no intended maximum size for other message types. The default value in the stream head is INFPSZ, the maximum that STREAMS allows.
SO_HIWAT	Set the high water mark value in the stream head read-queue to the value specified in the so_hiwat field of the stroptions structure.
SO_LOWAT	Set the low water mark value in the stream head read-queue to the value specified in the so_lowat field of the stroptions structure.
SO_BAND	Use in conjunction with the SO_HIWAT and SO_LOWAT flags, to set the high and low water marks for the priority band (in the stream head read-queue) specified by the so_band field in the stroptions structure. If the SO_BAND flag is not set, and the SO_HIWAT and SO_LOWAT flags are on, the normal high and low water marks are affected. The SO_BAND flag has no effect if SO_HIWAT and SO_LOWAT flags are off. Only one band's water marks can be updated with a single M_SETOPTS message.
SO_MREADON	Enable the stream head to generate an M_READ message when processing a *read* () system call. If both SO_MREADON and SO_MREADOFF are set in so_flags, SO_MREADOFF will have precedence.
SO_MREADOFF	Disable the stream head from generating an M_READ message when processing a read() system call. This is the default. If both SO_MREADON and SO_MREADOFF are set in so_flags, SO_MREADOFF flag takes precedence.

SO_ISTTY	Notify the stream head that the stream is acting as a controlling terminal.
SO_ISNTTY	Notify the stream head that the stream is no longer a controlling terminal. For SO_ISTTY, the stream may or may not be allocated as a controlling terminal via an M_SETOPTS message arriving upstream during open processing. If the stream head is opened before receiving this message, the stream will not be allocated as a controlling terminal until it is queued again by a session leader.
SO_NDELON	Set non-STREAMS tty semantics for O_NDELAY (or O_NONBLOCK) processing on *read* (2) and *write* (2) system calls. If O_NDELAY (or O_NONBLOCK) is set, a *read* (2) will return 0 if no data is waiting to be read at the stream head. If O_NDELAY (or O_NONBLOCK) is clear, a *read* (2) will block until data become available at the stream head.What is tty? Regardless of the state of O_NDELAY (or O_NONBLOCK), a *write* (2) will block on flow-control and will block if buffers are not available. If both SO_NDELON and SO_NDELOFF are set in the so_flags, SO_NDELOFF will have precedence.
SO_NDELOFF	Set STREAMS semantics for O_NDELAY (or O_NONBLOCK) processing on *read* (2) and *write* (2) system calls. If O_NDELAY (or O_NONBLOCK) is set, a *read* (2) will return -1 and set EAGAIN if no data is waiting to be read at the stream head. If O_NDELAY (or O_NONBLOCK) is clear, a *read* (2) will block until data is available at the stream head. If O_NDELAY (or O_NONBLOCK) is set, a *write* (2) will return -1 and set EAGAIN if flow-control is in effect when the call is received. It will block if buffers are not available. If O_NDELAY (or O_NONBLOCK) is set, part of the buffer has been written, and flow-control or buffers not available condition is encountered, *write* (2) will terminate and return the number of bytes written. If O_NDELAY (or O_NONBLOCK) is clear, a *write* (2) will block on flow-control and will block if buffers are not available. This is the default. If both SO_NDELON and SO_NDELOFF are set in the so_flags, SO_NDELOFF will have precedence. In the STREAMS-based pipe mechanism, the behavior of *read* (2) and *write* (2) is different for the O_NDELAY and O_NONBLOCK flags. See *read* (2) and *write* (2) for details.
SO_TOSTOP	Notify the stream head to stop on background writes.
SO_TONSTOP	Notify the stream head not to stop on background writes. The SO_TOSTOP and SO_TONSTOP are used in conjunction with job control.

M_SIG

This message is generated by modules and drivers to post a signal to a process. On receipt of this message, the stream head evaluates the first data byte of the message as a signal number, defined in <sys/signal.h>. The associated signal is sent to the process(es) under following conditions:

- If the signal is SIGPOLL, it is sent only to those processes that have explicitly registered to receive the signal (see I_SETSIG in *streamio* (7)).
- If the signal is not SIGPOLL and the stream containing the sending module or driver is a controlling tty, the signal is sent to the associated process group. A stream becomes the controlling tty for its process group if, on open(), a module or driver sends an M_SETOPTS message to the stream head with the SO_ISTTY flag set.

- If the signal is not `SIGPOLL` and the stream is not a controlling tty, no signal is sent, except in case of `SIOCSPGRP` and `TIOCSPGRP`. These two IOCTL commands set the process group field in the stream head so the stream can generate signals even if it is not a controlling tty.

M_TRAIL

This message is generated and sent upstream by a driver after the M_HANGUP message. It marks the end of data after an M_HANGUP message. The M_TRAIL message will be processed at the stream head only if the preceding M_HANGUP message sent by the driver contains the 2-byte TRAIL_TOKEN.

- Once the M_TRAIL message is processed, subsequent read() or getmsg() calls to the stream will return any messages present at the stream head.
- After all the messages are read, read() will return 0 and getmsg() will set each of its two length fields to 0.

High Priority Messages

M_CLOSE

This message is generated by the stream head to notify driver of a *close* (2) when the driver specifies C_ALLCLOSES in the d_flags field of drv_ops_t structure. When the C_ALLCLOSES flag is set, STREAMS sends an M_CLOSE message downstream when a close is issued on the driver. It then waits for a reply from the driver. The stream will be dismantled if the reply (M_CLOSE_REPL message) for the M_CLOSE message has MSGLASTCLOSE set in the b_flag field. If the M_CLOSE_REPL message does not have MSGLASTCLOSE set in the b_flag field, STREAMS will return without dismantling the stream.

M_CLOSE_REPL

This message is generated by the driver, in response to the M_CLOSE message sent to it by the stream head. When the flag C_ALLCLOSES is set in the d_flags field of the drv_ops_t structure, it is the responsibility of the driver to keep track of the information about the number of opens/closes for this device. Accordingly, for the last close on the device, the driver should set the MSGLASTCLOSE flag in the b_flag field of the M_CLOSE_REPL message, if it wants the stream associated with device to be dismantled.

M_COPYIN

The M_COPYIN message is generated by modules or drivers to request a copyin() to be performed on their behalf by stream head. This message is valid only after the M_IOCTL message is received by a module or driver and before an M_IOCACK or M_IOCNAK message is sent upstream by the module or driver. The format of a M_COPYIN message is one message block of type M_COPYIN, whose data buffer contains a copyreq structure defined in <sys/stream.h> and is shown here:

```
int          cq_cmd;         /* command type == ioc_cmd */
cred_t *     cq_cr;          /* pointer to full credentials */
uint         cq_id;          /* ioctl id == ioc_id */
int          cq_flag;        /* reserved */
mblk_t *     cq_private;     /* module's private state info */
ioc_pad      cq_ad;          /* address to copy data to/from */
uint         cq_size;        /* number of bytes to copy */
```

The first four fields of the copyreq structure correspond to those of the iocblk structure in the M_IOCTL message. This allows the same message block to be reused for M_COPYIN and M_IOCTL structures. The stream head guarantees that the message block allocated for M_IOCTL is large enough to contain a copyreq structure. The cq_ad refers to the address of the data buffer from which data needs to be copied for an M_COPYIN message. The cq_size specifies the number of bytes to be copied. Both cq_ad and cq_size values need to be set by the modules. The cq_flag field is reserved for future use and should be set to zero. cq_private can be set by the modules and drivers to get their state information pertaining to this IOCTL. The stream head will copy the contents of cq_private field to the M_IOCDATA response message so that the module can resume the associated state. If an M_COPYIN or M_COPYOUT message is freed, STREAMS will not free any message block pointed to by cq_private. It is then the modules responsibility to free any associated data in cq_private. This message should not be queued by a module or driver unless it intends to process the data for the IOCTL.

M_COPYOUT

The M_COPYOUT message is generated by modules or drivers to request a copyout() to be performed on their behalf by stream head. This message is valid only after receiving the M_IOCTL message by a module or driver and before an M_IOCACK or M_IOCNAK message is sent upstream by a module or driver. The format of a M_COPYOUT message is one message block of M_COPYOUT linked to one more M_DATA blocks containing data to be copied to the user's buffer. The data buffer in an M_COPYOUT message also contains a copyreq structure with the following differences:

- The cq_ad refers to the address of the buffer from which data needs to be copied out for an M_COPYOUT request.
- The cq_size specifies the number of bytes to be copied to the user space.

Data to be copied to the user space is contained in the linked M_DATA blocks. This message should not be queued by a module or driver unless it intends to process the data for the IOCTL in some way.

M_ERROR

This message is sent upstream by modules and drivers to report downstream error conditions. Upon receiving this message, the stream head does the following:

- Marks the stream in error state so that all subsequent system calls issued to the stream will fail (except the *close* (2) and *poll* (2) system calls).
- Sets errno to the first byte of the message. The POLLERR is set if the stream is being polled (see *poll* (2)).
- Awakens all processes sleeping on a system call to the stream.
- Sends an M_FLUSH message downstream with FLUSHRW set.

The stream head maintains an error field for the read-side and one for the write-side. The M_ERROR message could be in the one-byte or two-byte format. The one-byte format M_ERROR message sets both of the stream head error fields to the error specified by the first byte in the message. In the two-byte format, the first byte is used to set the read-side error and the second-byte is used to set the write-side error. If NOERROR is set in any of the two-bytes, the corresponding side of the stream is unchanged. This provides flexibility to modules and drivers to set different errors on the read-side and write-side, or set the error on only one side of the stream. If a byte is set to 0, the error state is cleared for the corresponding side of a stream. The values, NOERROR and 0 are not valid for the one-byte format of the M_ERROR message.

M_FLUSH

This message requests all modules and drivers that receive this message, to discard all messages in their corresponding message queues. An M_FLUSH can be sent by the stream head, or any module or driver. The first byte of the message contains flag bits that specifies the queues to be flushed. The flags bits are:

FLUSHR:	Flush the read queue of the module.
FLUSHW:	Flush the write queue of the module.
FLUSHRW:	Flush both the read queue and the write queue of the module.
FLUSHBAND:	Flush the message according to the priority associated with the band.

When a module receives an M_FLUSH message, it flushes appropriate queues and passes the message to the next stream component.

When a driver receives an M_FLUSH message, it does the following:

- If only FLUSHW is set, it flushes the write-queue and frees the message.

- If FLUSHR or FUSHRW is set, it flushes the read-queue, clears the FLUSHW flag and sends the message upstream.

When a stream head receives an M_FLUSH message, it does the following:

- If only FLUSHR is set, it flushes the read-queue and frees the message.
- If FLUSHW or FLUSHRW is set, it flushes the write-queue, turns the M_FLUSH message around, and sends it downstream.

If FLUSHBAND is set, the second byte of the M_FLUSH data buffer contains the band priority value. In this case, the flush flags apply only to the specified priority band.

NOTE All modules that enqueue messages must identify and process M_FLUSH messages.

M_HANGUP

This message is generated and sent upstream by a driver to indicate that the driver can no longer send data upstream. The reasons for generating this message are driver and device-dependent. For example, it could be an unrecoverable error or a remote line connection being dropped.

If a M_TRAIL message is used to mark the end of data after the M_HANGUP message, then the M_HANGUP message must contain the 2-byte TRAIL_TOKEN defined in <sys/stream.h>.

When the stream head receives a M_HANGUP message, it marks the stream in the hangup state, and all subsequent write() and putmsg() system calls issued to that stream will fail with an ENXIO error. Also, IOCTL commands requiring a message to be sent downstream will fail. The POLLHUP will be set if the stream is being polled (see *poll* (2)).

However, subsequent read() or getmsg() calls to the stream will not generate an error. These calls will return any messages that were in, or in transit to, the stream head read-queue before the M_HANGUP message was received. When all such messages have been read, read() will return 0 and getmsg() will set each of its two length fields to 0.

This message also causes a SIGHUP signal to be sent to the foreground process group if the device is a controlling terminal.

M_IOCACK

This message is used by modules and drivers to send a positive acknowledgment in response to an M_IOCTL message. The M_IOCACK message format is one M_IOCACK block (containing iocblk structure, see M_IOCTL) followed by zero or more M_DATA blocks. The ioc_rval field of the iocblk structure can be used to send return values to user process. ioc_error can be used to communicate processing errors, if any, back to the user in errno.

For M_IOCACK messages in response to I_STR IOCTL, the user data to be returned is formatted as an M_IOCACK message block followed by one or more M_DATA blocks that contain user data. The stream head returns the data to the user if there is a corresponding outstanding M_IOCTL request. Otherwise, the M_IOCACK message is ignored and all blocks in the message are freed.

Data can not be returned in an M_IOCACK message in response to a transparent IOCTL request. The user data to be returned for transparent IOCTL must be sent as M_COPYOUT messages. If any user data is linked as M_DATA blocks to the M_IOCACK block, the stream head will ignore and free them.

M_IOCNAK

This message is generated by a module and sent back to the stream as a negative acknowledgment of an M_IOCTL message. The M_IOCNAK message format is one M_IOCNAK block (containing iocblk structure, see M_IOCTL). The ioc_error field in iocblk structure can be used to communicate any processing errors, back to the user in errno. Unlike the M_IOCACK message, no user data or return values can be sent with the M_IOCNAK message. If any user data is linked as M_DATA blocks to the M_IOCNAK message, the stream head will ignore and free them. When the stream head receives an M_IOCNAK message, any outstanding IOCTL request will fail.

M_IOCDATA

This message is generated and sent downstream by the stream head in response to an M_COPYIN or M_COPYOUT message. The M_IOCDATA message is one message block of type M_IOCDATA, linked to zero or more M_DATA blocks. The data buffer of M_IOCDATA contains a copyresp structure defined as follows:

```
int            cp_cmd;        /* command type == ioc_cmd */
cred_t  *      cp_cr;         /* pointer to full credentials */
uint           cp_id;         /* ioctl id == ioc_id */
uint           cp_flag;       /* flag values */
mblk_t *       cp_private;    /* module's private state info */
ioc_pad        cp_rv;         /* 0 = success */
```

The first three fields of the copyresp structure correspond to those of the iocblk structure in the M_IOCTL message. This allows the same message block to be reused for all related transparent messages. The cp_rv field contains the result of the request at the stream head. Zero indicates success and non-zero indicates failure. If failure is indicated, the module must not generate an M_IOCNAK message. It must abort all IOCTL processing, clean up its data structures, and return. The cp_private field is copied from the cq_private field in the associated M_COPYIN/M_COPYOUT message. It is included in the M_IOCDATA message so the can be self-describing.

If the message is in response to an M_COPYIN message and success is indicated, the M_IOCDATA block will be followed by M_DATA blocks containing the copied data.

If an M_IOCDATA block is reused, any unused fields defined for the resultant message block should be cleared, especially for M_IOCACK and M_IOCNAK messages.

NOTE This message must not be queued by a module or driver unless it intends to process the data for the IOCTL in some way.

M_PCPROTO

This message type is similar to its non-priority counterpart M_PROTO and the following additional attributes:

- When an M_PCPROTO message is placed on a queue, its service procedure is always enabled.
- The stream head will allow only one M_PCPROTO message to be placed in its read queue at a time.
- If an M_PCPROTO message is already present on the stream head read queue, subsequent M_PCPROTO messages will be discarded and freed.

With M_PCPROTO, data and control information can be sent without any flow-control constraints. The getmsg() and putmsg() system calls refer to M_PCPROTO messages as high priority messages.

M_PCRSE

This is a priority message type reserved for internal use. Modules that do not recognize this message must pass it on. Drivers that do not recognize it must free it.

M_PCSIG

This message is similar to the `M_SIG` message type, except for the priority. `M_PCSIG` is generally preferred over `M_SIG`, as it is not subject to flow control.

M_READ

This message is generated by the stream head and sent downstream for a `read()` system call if no messages are waiting to be read at the stream head, and read notification has been enabled. The read notification can be enabled by `the SO_MREADON` flag, and disabled by the `SO_MREADOFF` flag of the `M_SETOPTS` message.

The message content is set to the number of bytes to be read in the `read()` call. This message notifies modules and drivers of a read, and supports communication between streams that reside on separate processors. The use of the `M_READ` message is developer dependent. Modules may either process this message and pass it on, or free the message when they recognize it. If modules do not recognize `an M_READ` message, they should pass it on. Drivers may or may not process it, and then free it. This message can not be generated by user processes and should not be generated by a module or driver. It is discarded if passed to the stream head.

M_START and M_STOP

This message requests devices to start or stop their output. They are intended to produce momentary pauses in a device's output, not to turn devices on or off. The message format and its use is developer dependent and can be considered to be special cases of the `M_CTL` message. These messages cannot be generated by user processes and are discarded if passed to the stream head.

M_STARTI and M_STOPI

These messages are similar to the `M_START` and `M_STOP` messages, except that `M_STARTI` and `M_STOPI` are used to start and stop input.

D STREAMS Administrative Driver

Overview

This appendix discusses the **STREAMS Administrative Driver** (SAD).

SAD

The **STREAMS Administrative Driver** (SAD) provides an interface to the `autopush` facility using the *ioctl* (2) function. As an interface, the `sad` driver enables administrative tasks to be performed on STREAMS modules and drivers. By specifying the command parameter to the *ioctl* (2) function, an administrator can configure `autopush` information for a device, get information on a device, or check a list of modules.

Synopsis

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

#include <sys/conf.h>

#include <sys/sad.h>

#include <sys/stropts.h>


int ioctl(int fd, int command, void *arg);
```

`fd`:	File descriptor obtained by opening `/dev/sad` via the *open* (2) system call.
`command`:	Administrative function to be performed. The supported commands are: `SAD_SAP`, `SAD_GAP`, `SAD_VML`.
`arg`:	Points to a `strapush` data structure if the IOCTL command is `SAD_SAP` or `SAD_GAP`, points to a `str_list` data structure if the IOCTL command is `SAD_VML`.

SAD_SAP

Allows the superuser to configure `autopush` information for a device. The *arg* parameter points to a `strapush` structure, defined in `<sys/sad.h>`, and shown here:

```
        struct strapush {

             uint sap_cmd;

             long sap_major;

             long sap_minor;

             long sap_lastminor;

             long sap_npush;

             char sap_list[MAXAPUSH][FMNAMESZ+1];

        };
```

`sap_cmd`: Allows the user to specify the type of configuration to perform. This field can have the following values:

`SAP_ALL`	Configures all minor devices.
`SAP_RANGE`	Configures a range of minor devices.
`SAP_ONE`	Configures a single minor device.

SAP_CLEAR	Clears the previous settings. Specifies only the sap_major and sap_minor fields when using this command. If a previous entry specified SAP_ALL, set the sap_minor field to 0 (zero). If a previous entry was specified as SAP_RANGE, set the sap_minor field to the lowest minor device number in the range.

sap_major:	Major device number.
sap_minor:	Minor device number.
sap_lastminor:	Range of minor devices.
sap_npush:	Number of modules to push. This number must be no more than **MAXAPUSH**, a constant defined in <sys/sad.h>. Additionally, this number must not exceed the STREAMS kernel tunable **NSTRPUSH**.
sap_list:	Specifies, in order, the array of modules to push.

SAD_GAP

Allows the sad driver to be used to obtain autopush configuration information for a device by setting the sap_major and sap_minor fields of the strapush structure (see the SAD_SAP command) to the major and minor device numbers of the device being queried. This strapush structure must be pointed to by the *arg* parameter. Upon successful completion, the strapush structure contains all of the information used to configure the device. Values of 0 (zero) will appear in any unused entry in the module list.

SAD_VML

This command enables the user to check a list of modules (e.g., to see if a specific module is installed). The *arg* parameter points to a str_list structure, defined in <sys/stropts.h> and shown here:

```
    struct str_list {
         int sl_nmods;
         struct str_mlist *sl_modlist;
   };


  sl_nmods: Number of entries allocated in an array.
  sl_modlist: Pointer to the array of module names.
```

The str_mlist structure, also defined in <sys/stropts.h>, contains the following:

```
struct str_mlist {
     char l_name[FMNAMESZ+1];
};
```

l_name:	Array of module names. If the l_name array is valid, the SAD_VML command returns a value of 0 (zero). If the array contains an invalid module name, the command returns a value of 1. Upon failure, the command returns a value of -1.

NOTE As a STREAMS driver, sad also supports the normal STREAMS I_STR IOCTL command. In this form, the developer needs to set the ic_cmd field in the strioctl structure to either SAD_SAP, SAD_GAP, or SAD_VML. The ic_dp field of the strioctl structure should point to this strapush structure.

Return Value Unless specified otherwise, upon successful completion, the `sad` IOCTL commands return a value of `0` (zero). Otherwise, a value of `-1` is returned.

Errors If any of the following conditions occur, the `sad` IOCTL commands return the corresponding value:

SAD_SAP

[EEXIST]	The specified major/minor device number pair (`sad_major`/`sad_minor`) has already been configured.
[EFAULT]	The *arg* parameter points outside the allocated address space.
[EINVAL]	The major device number (`sad_major`) is invalid, the number of modules (`sap_list [MAXAPUSH][FMNAMESZ+1]`) is invalid, or the list of module names is invalid.
[ENODEV]	The device is not configured for `autopush`. This value is returned from a `SAD_GAP` command.
[ENOSR]	A internal `autopush` data structure cannot be allocated.
[ENOSTR]	The major device does not represent a STREAMS driver.
[ERANGE]	The `sap_lastminor` field is less than the `sap_minor` field when the command is `SAP_RANGE`, or the minor device specified in a `SAP_CLEAR` command does not exist.
[EACCES]	Only superuser is allowed to execute the `SAD_SAP` IOCTL.

SAD_GAP

[EFAULT]	The *arg* parameter points outside the allocated address space.
[EINVAL]	The major device number (`sad_major`) is invalid.
[ENODEV]	The device is not configured for `autopush`.
[ENOSTR]	The major device does not represent a STREAMS driver.

SAD_VML

[EFAULT]	The *arg* parameter points outside the allocated address space.
[EINVAL]	The list of module names is invalid.

E Differences Between STREAMS/UX and System V Release 4 STREAMS

This appendix summarizes the differences between STREAMS/UX and System V Release 4.2 STREAMS.

This appendix will be divided into the following categories for describing differences between HP-UX and SVR4.2 STREAMS:

- Commands
- System calls
- Utilities
- Drivers and modules
- Data structures
- Message types
- Cloning
- Hardware driver writing

HP-UX Changes to STREAMS Commands

STREAMS/UX supports the commands listed below:

- `autopush`
- `fdetach`
- `strace`
- `strchg`
- `strclean`
- `strconf`
- `strerr`
- `strvf`

HP versions of supported STREAMS/UX commands operate somewhat differently from the way the commands are described in the *UNIX SVR4.2 Command Reference* manual. NLS catalogs exist for the commands. The catalogs are called `autopush.cat`, `fdetach.cat`, `strace.cat`, `strchg.cat`, `strclean.cat`, `strconf.cat`, `strerr.cat`, and `strvf.cat` and are located in the `/usr/lib/nls/C` directory. Differences in the commands are described:

autopush

The syntax for the `autopush` command on HP-UX is as follows:

```
autopush   -f   autopush_file_name

autopush   -r   -M   major_num|dev_name   -m   minor_num

autopush   -g   -M   major_num|dev_name   -m   minor_num
```

`autopush_file_name` contents:

```
major_num|dev_name low_minor high_minor mod_name 1...mod_name N
```

The HP-UX `autopush` command has been enhanced to allow the user to specify the device name in place of the major number, which is recommended since HP-UX provides dynamic major numbers. The name can be specified in the `autopush` file and on the command line. The major number can still be used if needed.

strace and strerr

The `strace` and `strerr` commands use the STREAMS log driver, `/dev/strlog`. SVR4.2 calls this driver `/dev/log`. Since HP-UX already includes a non-streams driver named `/dev/log` the streams log driver has been renamed as `/dev/strlog`.

HP-UX Changes to STREAMS/UX System Calls

STREAMS/UX supports the following system calls:

- `close`
- `fattach`
- `fcntl`
- `fdetach`
- `getmsg`
- `getpmsg`
- `ioctl`
- `isastream`
- `open`
- `pipe`
- `poll`
- `putmsg`
- `putpmsg`
- `read`
- `readv`
- `select`
- `signal`
- `write`
- `writev`

For STREAMS-based `termio`, see the following manpages (which are part of the STREAMS-TIO product): *grantpt* (3C), *ptsname* (3C), and *unlockpt* (3C).

There are HP-UX modifications to the `fattach`, `ioctl`, `pipe`, `poll`, `putmsg`, `putpmsg`, `select`, `signal`, `write`, and `writev` system calls. These modifications are as follows.

fattach/fdetach Modifications

STREAMS/UX supports the *fattach* (3) and *fdetach* (3) library calls and the *fdetach* (1m) command as described in the *UNIX SVR4.2 Operating System API Reference* and the *SVR4.2 Command Reference*. In order to use `fattach` and `fdetach`, the kernel must have the ffs file system configured. If ffs has been deleted, reinclude ffs in the system file using the HP-UX kernel configuration command `kcmodule`, regenerate a kernel and reboot the system. For more information about how to regenerate the HP-UX kernel, refer to the *HP-UX Driver Development Guid*e.

ioctl Modifications

STREAMS supports IOCTL as described in the *SVR4.2 STREAMS* manual.

NOTE The multiplexor ID number returned by `I_LINK` and `I_PLINK` is opaque to the user and not a small integer such as 0, 1, 2, 3.

pipe Modifications

STREAMS/UX supports STREAMS-based pipes as an optional feature. STREAMS/UX's STREAMS-based pipes behave as described in the *UNIX SVR4.2 Operating System API Reference* and the *UNIX System V Release 4 Programmer's Guide: STREAMS*.

By default, pipes created by the *pipe* (2) system call are not STREAMS-based. In order to get STREAMS-based pipes, the `/stand/system` file must have the `pipemod` and `pipedev` module and driver configured, and the tunable parameter streampipes must be set to 1 (one).

When STREAMS/UX is installed, the `/stand/system` file is modified to include `pipemod` and `pipedev`, but streampipes is set to zero by default. The kernel must be regenerated and the system rebooted if the setting of streampipes to non-zero is to take effect. In other words, adb'ing the running system to turn streampipes on will have no effect on the type of pipes created by *pipe* (2). Once the kernel is regenerated and rebooted, all *pipe* (2) pipes on the system will be STREAMS-based. However, FIFOs will not be STREAMS-based. STREAMS/UX does not support STREAMS-based FIFOs.

The STREAMS/UX device `pipedev` is only for internal STREAMS/UX use in implementing STREAMS-based pipes. Opening a device file with `pipedev`'s major number will not result in a STREAMS-based pipe, or even a properly functioning stream. STREAMS-based pipes must be created using the *pipe* (2) system call.

The `PIPE_BUF` is a pathname variable value, and SVID, XPG4, POSIX, etc. define it as the maximum number of bytes that is guaranteed to be written atomically. To obtain the correct value of `PIPE_BUF`, use `fpathconf()` (see `pathconf()`). For STREAMS-based pipes, the value of `PIPE_BUF` depends on the configurable parameter `STRMSGSZ` (by default, 8KB). For example, `PIPE_BUF` is set to 4KB if `STRMSGSZ` is 4KB, 8KB if `STRMSGSZ` is 8KB, and 16KB if `STRMSGSZ` is 16KB. There is one exception. If `STRMSGSZ` is set to 0, then `PIPE_BUF` for STREAMS/UX pipes is set to 8KB.

putmsg and putpmsg Modifications

Maximum and Minimum Data Buffer Size:

The size of the user's data buffer must be within the minimum and maximum packet size range specified in the topmost STREAM module's streamtab. It must also be less than or equal to `STRMSGSZ`. If the number of bytes to transfer is not in this range, `ERANGE` will be returned.

Maximum and Minimum Control Buffer Size:

The size of the user's control buffer must be less than or equal to both `STRCTLSZ` and `STRMSGSZ`. If `STRCTLSZ` is less than or equal to zero, the page size is used instead of `STRCTLSZ` for this check.

Data Buffer Segmentation

The user's data buffer may be sent in multiple data blocks chained together to form a message. The maximum number of bytes, including the write offset, that can be sent in one data block is equal to the page size.

Write Offset

A module or driver can send the stream head an `M_SETOPTS` message, telling the STREAM head to put an offset in the beginning of the first data block in a message sent by a `putmsg` call. STREAMS/UX will not put the offset into the data block if the amount of memory required is greater than the page size.

select Modifications

STREAMS/UX supports the select system call for STREAMS devices. For more information about the `select` system call, see the *select* (2) manpage provided with the HP-UX system.

The `select` system call does not provide as much information as `poll`. If `select` returns an event for a STREAMS device, the program can call `poll` to get more information.

A select read event is returned if a poll event `POLLRDNORM`, `POLLERR`, `POLLNVAL` or `POLLHUP` exists on the stream. In other words, a read event is returned for the following conditions:

- Normal message is waiting to be read
- Read error exists at the stream head
- Write error exists at the stream head
- Stream is linked under a multiplexor
- Hang-up has occurred

A select write event is returned if a `poll` event `POLLOUT`, `POLLWRNORM`, `POLLERR`, `POLLNVAL`, or `POLLHUP` exists on the STREAM. This means that a write event is returned for the following conditions:

- Normal data can be written without blocking because of flow control
- Read error exists at the stream head
- Write error exists at the stream head
- Stream is linked under a multiplexor
- Hang-up has occurred

A select exception event is returned if a poll event `POLLPRI` or `POLLRDBAND` exists on the STREAM. More specifically, an exception event is returned if a high-priority message or a banded message is waiting to be read.

/dev/poll Interface

HP-UX offers the `/dev/poll` interface as an added feature and as alternate functionality to *poll* (2). It provides an interface to the event port driver allowing a user to synchronously monitor a specific set of conditions associated with the registered set of file descriptors. Access to `/dev/poll` is provided through the *open* (2), *write* (2), and *ioctl* (2) system calls.

The `/dev/poll` event port provides functionality comparable to the *select* (2) and *poll* (2) system calls. General operations supported by the event port driver are:

- Opening an event port
- Registering and deregistering file descriptors on an event port
- Polling registered file descriptors on an event port
- Retrieving registered poll conditions for a file descriptor

- Closing an event port

The *poll* (7) manpages provide more information on this interface.

signal Modifications

STREAMS/UX supports signals and the HP-UX signal system call. However, STREAMS/UX does not support extended signals or the `siginfo_t` structure described in the *siginfo* (5) manpage.

write and writev Modifications

Maximum and Minimum Data Buffer Size

The size of the user's data buffer must be within the minimum and maximum packet size range specified in the topmost STREAM module's streamtab. If the number of bytes to transfer is not in this range, `ERANGE` will be returned. Two exceptions exist in which no error occurs. The first exception is if the data buffer is too large and either the maximum packet size is infinite or the minimum packet size is less than or equal to zero. (An infinite packet size is specified using the define `INFPSZ` in the `stream.h` file.) The second exception occurs if the buffer is too small and the minimum packet size is less than or equal to zero. With either exception, `ERANGE` is not returned, and the data is transferred.

Data Buffer Segmentation

The user's data buffer may be sent in multiple messages. The maximum amount of data that can be sent in one message is the lower value of the topmost module's maximum packet size and `STRMSGSZ`. If the maximum packet size is infinite, then the top module's high water mark is taken into consideration. If the high water mark is more than zero, half of the high water mark is used; otherwise the page size is used.

Write Offset

A module or driver can send the STREAM head an `M_SETOPTS` message telling it to put an offset in the beginning of each data buffer segment (i.e., message) sent by a write call. STREAMS/UX will not put the offset into a message if the resulting message size exceeds `STRMSGSZ`.

HP-UX Modifications to STREAMS/UX Utilities

STREAMS/UX supports the following kernel utilities described in the *SVR4.2 Driver* manual, although some of the utilities have been modified for HP-UX.

`adjmsg`	`freeb`	`noenable`	`strlog`
`allocb`	`freemsg`	`OTHERQ`	`strqget`
`bacvkq`	`freezestr`	`pcmsg`	`strqset`
`bcanput`	`getadmin`	`pullupmsg`	`SV_ALLOC`
`bcanputnext`	`getmid`	`put`	`SV_BROADCAST`
`bcopy`	`getmajor`	`putbq`	`SV_DEALLOC`
`bufcall`	`getminor`	`putctl`	`SV_WAIT`
`bzero`	`getq`	`putctl1`	`SV_WAIT_SIG`
`canput`	`insq`	`putnext`	`testb`
`canputnext`	`itimeout`	`putnextctl`	`timeout`
`cmn_err`	`kmen_alloc`	`putnextctl1`	`TRYLOCK`
`copyb`	`kmem_free`	`putq`	`unbufcall`
`copymsg`	`linkb`	`qenable`	`unfreezestr`
`datamsg`	`LOCK`	`qprocsoff`	`unlinkb`
`delay`	`LOCK_ALLOC`	`qprocson`	`UNLOCK`
`drv_getparm`	`LOCK_DEALLOC`	`qreply`	`untimeout`
`drv_priv`	`major`	`qsize`	`vtop`
`dupb`	`makedev`	`RD`	`wakeup`
`dupmsg`	`makedevice`	`rmvb`	`WR`
`enableok`	`max`	`rmvq`	
`esballoc`	`min`	`SAMESTR`	
`esbbcall`	`minor`	`sleep`	
`flushband`	`msgdsize`	`spln`	
`flushq`	`msgppullup`	`splstr`	

In addition, HP-UX provides the following new utilities.

```
get_sleep_lock
putctl2
putnextctl2
streams_put
```

`unweldq`

`weldq`

The `strenv.h` file redefines some native HP-UX kernel utilities to conform to System V Release 4.2. The `strenv.h` file redefines delay, `get_sleep_lock`, `kmem_alloc`, `kmem_free`, `lbolt`, `max`, `min`, `sleep`, `time`, `timeout`, and `untimeout`. These defines might collide with declarations in STREAMS/UX modules and drivers. The `strenv.h` file can be customized to avoid collisions or to use native HP-UX utilities. However, modules and drivers cannot call the native HP-UX `sleep` or `get_sleep_lock` directly. If modules and drivers call `sleep` or `get_sleep_lock`, they must include `strenv.h` to redefine `sleep` and `get_sleep_lock` to `streams_mpsleep` and `streams_get_sleep_lock`.

Differences between the STREAMS/UX kernel utilities and the descriptions in the *SVR4.2 Driver* manual are discussed here, along with information about new utilities. This section assumes that modules and drivers include `strenv.h`.

cmn_err

The STREAMS/UX `cmn_err` is the same as the `cmn_err` described in the *SVR4.2 Driver* manual with a few differences. The HP-UX `cmn_err` always sends messages to both the system console and the circular kernel buffer. Inserting an exclamation point (“!”) or a circumflex (“^”) as the first character in the format string has no effect. HP-UX simply removes these control characters from the message, and sends the message to both the console and the kernel buffer. There are a couple of other very minor differences. HP-UX precedes `CE_PANIC` level messages with the string `panic:` instead of `PANIC:`. Also, the HP-UX circular kernel buffer is called **msgbuf** instead of **putbuf**. The HP-UX `msgbuf` is a fixed size, and can be viewed using the `dmesg` command or the KWDB debugger tool.

esballoc

The STREAMS/UX `esballoc` is the same as the `esballoc` call described in the *SVR4.2 Driver* manual with a few differences. The HP-UX `esballoc` copies the contents of the `fr_rtn` structure into an area of the message block not visible to the STREAMS/UX programmer. The free routine passed to `esballoc` can call STREAMS/UX utilities in the same way as the put or service procedure that called `freeb`. Also, a free routine can safely access the same data structures as the put or service routine that called `freeb`. However, unlike SVR4.2, HP-UX does not block interrupts from all STREAMS/UX devices while the free routine runs.

freezestr and unfreezestr

The *SVR4.2 Driver* manual says that `freezestr` and `unfreezestr` must be called on multiprocessors to protect searching a STREAMS/UX queue and calling `insq`, `rmvq`, `strqset`, and `strqget`. SVR4 MP provides `freezestr` and `unfreezestr` to prevent software on multiple processors from manipulating a queue’s list of messages at the same time. STREAMS/UX uses synchronization levels for this. Appendix-F for more information about synchronization levels. STREAMS/UX uses a different mechanism to protect STREAMS/UX queues. Thus the HP-UX freezestr/unfreezestr mechanisms are different from SVR4.2 definitions in that they freeze/unfreeze the state of a single queue and not the state of the entire stream containing the queue.

All modules/drivers with the synchronization level `SQLVL_NOSYNC` still need to call `freezestr` to freeze the queue. The call to `freezestr` needs to be done before calling `insq`, `rmvq`, `strqget`, and `strqset` to prevent multiple `put` routines running on different processors from manipulating the same queue. A call to `unfreezestr` must be made to unfreeze the queue.

For module/drivers with a synchronization level other than `SQLVL_NOSYNC`, `freezestr` returns the current interrupt priority level, and `unfreezestr` is a no-op. HP-UX provides the `freezestr` and `unfreezestr` stubs for easier porting of code from SVR4 MP, for these levels.

get_sleep_lock

STREAMS/UX provides some extra support for modules and drivers which use the native HP-UX `get_sleep_lock` primitive. Alternatively, modules and drivers can call the SVR4 MP `SV_WAIT` and `SV_WAIT_SIG`. Open and close routines call `get_sleep_lock` before sleeping to prevent missing wakeups. After calling `get_sleep_lock`, the open or close can release spinlocks before sleeping. Other processes cannot wakeup the open or close between the time it calls `get_sleep_lock` and sleep. Modules and drivers must include `strenv.h` to use `get_sleep_lock`. `strenv.h` redefines `get_sleep_lock` to `streams_get_sleep_lock`. Modules and drivers cannot call the native HP-UX `get_sleep_lock` directly, because STREAMS/UX needs to do some additional synchronization before invoking `get_sleep_lock`.

```
lock_t * get_sleep_lock(caddr_t event);
```

The open or close routine passes the event it will pass to the sleep primitive to `get_sleep_lock`. `get_sleep_lock` obtains a sleep spinlock, and returns a pointer to this lock.

itimeout

If the HP-UX `itimeout` cannot allocate memory, it panics instead of returning 0 like the SVR4 MP `itimeout`. The STREAMS/UX `itimeout` only returns 0 if it is passed an interrupt priority level that is lower than `pltimeout`.

kmem_alloc

A value of 0 for the size parameter is illegal in STREAMS/UX `kmem_alloc()`. The SVR4.2 `kmem_alloc` returns NULL instead.

LOCK

The STREAMS/UX LOCK calls the native HP-UX spinlock primitive. LOCK has an interrupt priority level parameter, which is used to raise the priority level and block interrupts which acquire the spinlock. The *SVR4.2 Driver* manual says that implementations which do not need to raise the interrupt level can ignore this parameter. Since the HP-UX spinlock primitive always raises the interrupt level to `spl6` while a spinlock is held, STREAMS/UX ignores the interrupt level parameter on multiprocessor systems. For better performance on uniprocessor systems, the STREAMS/UX LOCK raises the priority level to the parameter value instead of acquiring a spinlock. Whether the caller will block or spin if the lock cannot be obtained is implementation defined. The HP-UX implementation spins.

LOCK_ALLOC

The STREAMS/UX `LOCK_ALLOC` calls the native HP-UX `alloc_spinlock` primitive. There are some small differences between the STREAMS/UX `LOCK_ALLOC` and the SVR4 MP utility. The `LOCK_ALLOC` has a flag parameter which indicates if the caller is willing to block while waiting for memory to be allocated. HP-UX only allows this flag to be set to `KM_SLEEP`, and returns zero if it is set to `KM_NOSLEEP`. The STREAMS/UX `LOCK_ALLOC` accepts the following hierarchy parameter values which are reserved for STREAMS/UX modules and drivers in `/usr/include/sys/semglobal.h`:

- `STREAMS_USR1_LOCK_ORDER`
- `STREAMS_USR2_LOCK_ORDER`
- `STREAMS_USR3_LOCK_ORDER`

The compiler options to turn on deadlock checking for HP-UX are different than those documented in the *SVR4.2 Driver* manual. The entire HP-UX kernel and the module or driver must be compiled with `SEMAPHORE_DEBUG` to enable deadlock checking. According to the *SVR4.2 Driver* manual, the *`min_pl`* parameter can be ignored by implementations which do not need to raise the priority level. The HP-UX `STREAMS LOCK_ALLOC` ignores it.

putctl2

STREAMS/UX also provides the additional utility called `putctl2`. This utility can be used to send a control message with a two-byte parameter to a queue. For example, `putctl2` can send the new style of an `M_ERROR` message, which is two bytes long, to the specified queue:

```
int putctl2(queue_t *q, int type, int p1, int p2);
```

where,

`q`	The queue to which the message is sent
`type`	The message type
`p1,p2`	Two bytes of data in the message.

The `putctl2` utility ensures that the type is not a data type. The utility also allocates a message block, fills in the data, and calls the put procedure of the specified queue.

The `putctl2` utility returns 0 if the type is `M_DATA`, `M_PROTO` or `M_PCPROTO`, or if a message block cannot be allocated. `putctl2` utility returns 1 if it completes successfully.

putnextctl2

STREAMS/UX provides an additional utility `putnextctl2`. This utility can be used to send a control message with a two-byte parameter to the next queue in a stream. For example, `putnextctl2` can send the new style of an `M_ERROR` message, which is two bytes long, to the next queue in a stream:

```
int putnextctl2(queue_t *q, int type, int p1, int p2);
```

where,

`q`	The queue from which the message is sent to `q->q_next`
`type`	is the message type
`p1,p2`	Two bytes of data in the message.

The `putnextctl2` utility ensures that the type is not a data type. The utility also allocates a message block, fills in the data, and calls the put procedure of `q->q_next`.

The `putnextctl2` utility returns 0 if the type is `M_DATA`, `M_PROTO`, or `M_PCPROTO`, or if a message block cannot be allocated. `putnextctl2` utility returns 1 if it completes successfully.

qprocson and qprocsoff

SVR4 MPSTREAMS provides `qprocson` and `qprocsoff`, which on a multiprocessor system allows a module's put and service procedures to run concurrently with open and close. STREAMS/UX does not allow this parallelism. A module's or driver's put and service procedures cannot run at the same time as the open or close. Although STREAMS/UX does not run the put or service procedure in parallel with the open or close, it does queue any requests to run the put or service procedure. STREAMS/UX will process these when open finishes. Also, if open or close sleeps, STREAMS/UX can run the put and service procedures while open or close are sleeping. However, a put or service procedure cannot do the wakeup on a sleeping open or close. STREAMS/UX provides stubs which are no-ops for `qprocson` and `qprocsoff` to make porting easier.

streams_put utility

STREAMS/UX provides a new utility `streams_put()`, which allows non-STREAMS software to safely call STREAMS/UX utilities. `timeout` and `bufcall` user functions and other non-STREAMS code cannot call several of the STREAMS utilities or share data with modules and drivers.

Non-STREAMS code can call `streams_put()`, passing it a function and a queue. STREAMS/UX runs the function as if it were the queue's put procedure. The function can safely manipulate the queue and access the same data structures as the queue's put procedure.

```
typedef void (*streams_put_t)(void *, MBPKP);
```

```
void streams_put(streams_put_t func, queue_t *q, mblk_t *mp, void *private);
```

STREAMS will run `func` as if it were *q*'s put procedure. STREAMS passes `private` and `mp` to `func`. The non-STREAMS code can pass any value in the private parameter. The code must pass a valid message block pointer in mp.

SV_WAIT

STREAMS/UX implements a subset of the SVR4 MP synchronization variable utilities using sleep and wakeup. The STREAMS/UX `SV_WAIT` differs from the SVR4 MP utility in the following ways.

- When the SVR4 MP `SV_WAIT` returns, the `lkp` spinlock is not held and the priority level is set to `plbase` (`SPLNOPREEMPT` on HP-UX). On a multiprocessor system, the HP-UX `SV_WAIT` lowers the priority level to the value before the caller acquired the `lkp` spinlock, which may not be `SPLNOPREEMPT`. If the caller acquired the lock while holding other spinlocks, the priority level is lowered to the value before the first of these nested spinlock calls.
- The SVR4 MP `SV_WAIT` has a priority argument that specifies the priority the caller would like to run at after waking. Since the HP-UX `SV_WAIT` is implemented by calling sleep, the HP-UX priorities are different than the SVR4 MP ones. On HP-UX, the priority passed into `SV_WAIT` is subtracted from `PZERO-1`. The `pridisk`, `prinet`, `pritty`, `pritape`, `prihi`, `primed`, and `prilo` are defined to be 0 and do not affect the caller's priority. To change the process's priority, study the priorities in `/usr/include/sys/param.h` and pass the needed offset to `PZERO-1` in the priority parameter.

SV_WAIT_SIG

HP implements a subset of the SVR4 MP synchronization variable utilities using sleep and wakeup. The STREAMS/UX `SV_WAIT_SIG` differs from the SVR4 MP utility in the following ways.

- When the SVR4 MP `SV_WAIT_SIG` returns, the `lkp` spinlock is not held, and the priority level is set to `plbase` (`SPLNOPREEMPT` on HP-UX). On a multiprocessor system, the HP-UX `SV_WAIT_SIG` lowers the priority level to the value before the caller acquired the `lkp` spinlock, which may not be `SPLNOPREEMPT`.
- If the caller acquired the lock while holding other spinlocks, the priority level is lowered to the value before the first of these nested spinlock calls.
- Also, the SVR4 MP `SV_WAIT_SIG` has a priority argument that specifies the priority the caller would like to run at after waking. Since the HP-UX `SV_WAIT_SIG` is implemented by calling sleep, the HP-UX priorities are different than the SVR4 MP ones.
- On HP-UX, the priority passed into `SV_WAIT_SIG` is added to `PZERO+1|PCATCH`. `pridisk`, `prinet`, `pritty`, `pritape`, `prihi`, `primed` and `prilo` are defined to be 0 and do not affect the caller's priority. To change the process's priority, study the priorities in `/usr/include/sys/param.h` and pass the needed offset to `PZERO+1|PCATCH` in the priority parameter.

- The last difference is that the SVR4 MP `SV_WAIT_SIG` returns if the process is first stopped by a job control signal and then continued. The HP-UX `SV_WAIT_SIG` continues to sleep until it receives a signal which does not stop the process or an `SV_BROADCAST` wakes up the process.

TRYLOCK

The STREAMS/UX TRYLOCK calls the native HP-UX cspinlock primitive. TRYLOCK has an interrupt priority level parameter, which is used to raise the priority level and block interrupts which acquire the spinlock. The *SVR4.2 Driver* manual says that implementations which do not require the interrupt level to be raised, can ignore this parameter. STREAMS/UX ignores the parameter on multiprocessor systems since the HP-UX cspinlock primitive always raises the interrupt level to `spl6` while a spinlock is held. For better performance on uniprocessor systems, the STREAMS/UX TRYLOCK raises the priority level to the parameter value instead of acquiring a spinlock.

UNLOCK

The STREAMS/UX UNLOCK calls the native HP-UX spinunlock primitive. UNLOCK has an interrupt priority level parameter, which is used to lower the priority level. HP-UX will ignore this parameter on multiprocessor systems. If the caller is not holding any other spinlocks, the STREAMS/UX UNLOCK lowers the priority level to the value before the caller acquired the spinlock. On uniprocessor systems, the STREAMS/UX UNLOCK lowers the priority level to the parameter value instead of releasing a spinlock.

weldq and unweldq

STREAMS/UX provides the additional utilities `weldq` and `unweldq` to allow the user to build a pipe-like stream. These utilities are provided because the programmer is not allowed to modify `q_next` pointers directly

weldq

The `weldq` connects two drivers' queues to form a pipe by setting the `q_next` pointer:

```
 int weldq (queue_t *  d1_wq,
            queue_t *  d2_rq,
            queue_t *  d2_wq,
            queue_t *  d1_rq,
            weld_fcn_t func,
            weld_arg_t arg,
            queue_t *  protect_q);
```

`d1_wq, d1_rq`	One of the drivers' write and read queues.
`d2_wq, d2_rq`	The second driver's queues.
`func`	Callback function to be executed by the weld daemon.
`arg, protect_q`	Parameters passed to the previous function.

The `weldq` will set `d1_wq->q_next` to be `d2_rq` and `d2_wq->q_next` to `d1_rq`. Also, `weldq` updates internal queue fields used for flow control that are not visible to the STREAMS programmer.

The `weldq` returns to the caller before connecting the drivers. The `weldq` requests the STREAMS `weld` daemon to update the queues.

The weld daemon will call func with arg as an argument after it finishes the request. The protect_q specifies which queue the callback function can access safely. If the driver does not need to be notified when the daemon finishes the weld request, pass weldq zero for the func, arg and protect_q parameters.

On successful completion, weldq returns 0. However, if weldq fails, an errno indicating the type of error that has occurred is returned. The errno will contain one of the following three values:

ENXIO	The weld daemon is not running.
EINVAL	Invalid queue arguments exist.
EAGAIN	No memory is available.

unweldq

The utility unweldq disconnects two drivers' queues that were joined by weldq:.

```
 int unweldq (queue_t *  d1_wq,

              queue_t *  d2_rq,

              queue_t *  d2_wq,

              queue_t *  d1_rq,

              weld_fcn_t func,

              weld_arg_t arg,

              queue_t *  protect_q);
```

d1_wq, d1_rq	One of the drivers' write and read queues.
d2_wq, d2_rq	The second driver's queues.
func	Callback function to be executed by the weld daemon.
arg, protect_q	Parameters passed to the previous function.

The unweldq will set d1_wq->q_next and d2_wq->q_next to zero. Also, it updates internal queue fields used for flow control that are not visible to the STREAMS programmer.

The unweldq returns to the caller before disconnecting the drivers. The unweldq requests that the STREAMS weld daemon update the queues.

NOTE If one end of a pipe-like stream created by weld is closed, STREAMS will automatically unweld the two drivers. unweldq does not need to be called.

The weld daemon will call func with arg as an argument after it finishes the request. protect_q specifies which queue the callback function can access safely. If the driver does not need to be notified when the daemon finishes the weld request, pass weldq zero for the func, arg and protect_q parameters.

On successful completion, unweldq returns 0. Otherwise, it returns an errno indicating the type of error that occurred. One of the following four values will be returned:

ENXIO	The weld daemon is not running.
EINVAL	Invalid queue arguments are present.
EAGAIN	No memory is available.
ENXIO	The weld daemon is not running.

vtop

The STREAMS/UX `vtop` only accepts a NULL process structure pointer. In other words, it only converts kernel space addresses.

HP-UX Changes to STREAMS/UX Drivers and Modules

The unsupported drivers and modules include:

```
connld
console
ports
sxt
xt
```

NOTE Some STREAMS-based terminal I/O functionality is contained in a separate product called STREAMS-TIO. It is part of the HP-UX runtime product. See the following manpages (which are part of the STREAMS-TIO product): *pts* (7), *ptm* (7), *ldterm* (7), *pterm* (7) and *pckt* (7).

STREAMS/UX provides the following drivers and modules:

```
clone
strlog
sad
echo
sc
timod
tirdwr
pipemod
```

Entries for these drivers and modules can be found in the system file. General information about these drivers follows. Information about the stream head is also included. Differences between the HP-UX and SVR4.2 log and `sad` drivers are also described.

NOTE Any driver or module not explicitly listed as supported in this section is not supported.

clone

Major Number:	72

The `clone` is used to provide cloning. The major number of the device file for a cloneable driver must be the `clone` driver's major number, 72. The minor number is set to the real major number of the device.

strlog

Major Number:	73
Module ID Number:	44
Maximum Packet Size:	`INFPSZ`
Minimum Packet Size:	0
High Water Mark:	2048
Low Water Mark:	128

The STREAMS/UX log driver is named `strlog` instead of log. The special device file is `/dev/strlog`. `strlog` provides the same functionality for logging as described in the *UNIX SVR4.2 System Files and Devices Reference*, with the exceptions described:

- The `strlog` kernel utility formats binary arguments before sending messages up the stream.
- STREAMS/UX does not provide a separate console logger or `/dev/console` device. The `strlog` does not support the `I_CONSLOG` IOCTL. The `strlog` prints a log message on the console if the `SL_CONSOLE` flag is set.
- The HP-UX `log_ctl` structure does not contain a `pri` field. Priority and facility codes are not supported.

sad

Major Number:	74
Module ID Number:	45
Maximum Packet Size:	`INFPSZ`
Minimum Packet Size:	0
High Water Mark:	2048
Low Water Mark:	128

The STREAMS/UX `sad` driver device file is `/dev/sad`. The system administrator and users can open `/dev/sad`. However, only the system administrator can execute the `SAD_SAP` IOCTL system call. This differs from the System V `sad` driver, which is accessed through the `/dev/sad/admin` and `/dev/sad/user` device files.

The `sad` provides `autopush` functionality as described in the *UNIX SVR4.2 System Files and Devices Reference* manual.

echo

Major Number:	116
Module ID Number:	5000
Maximum Packet Size:	`INFPSZ`
Minimum Packet Size:	0
High Water Mark:	65536
Low Water Mark:	1024

The `echo` is a loopback driver used by the `strvf` STREAMS/UX verification tool. For more information about `strvf` see the manpage *strvf* (1M).

sc

Module ID Number:	5002
Maximum Packet Size:	`INFPSZ`
Minimum Packet Size:	0
High Water Mark:	2048
Low Water Mark:	128

The `sc` provides auxiliary functions for the `sad` driver.

timod

Module ID Number:	5006
Maximum Packet Size:	`INFPSZ`
Minimum Packet Size:	0
High Water Mark:	2048
Low Water Mark:	128

The `timod` provides TLI functionality as described in the *UNIX SVR4.2 System Files and Devices Reference* manual.

tirdwr

Module ID Number:	0
Maximum Packet Size:	`INFPSZ`
Minimum Packet Size:	0
High Water Mark:	16384
Low Water Mark:	128

The `tirdwr` provides an alternative interface to the TLI library for accessing a transport protocol provider. The `tirdwr` is described in the *UNIX SVR4.2 System Files and Devices Reference* manual.

Stream Head

Module ID Number:	0
Module Name:	sth
Maximum Packet Size:	`INFPSZ`
Minimum Packet Size:	0
High Water Mark:	65536
Low Water Mark:	8192

The Stream head provides the interface between HP-UX system calls and STREAMS/UX utilities in the kernel. The Stream head is the first queue pair of every Stream and is involved in flow control. Data being read from a stream will be taken off the stream head.

pipemod

Module ID Number:	5303
Maximum Packet Size:	8192
Minimum Packet Size:	0
High Water Mark:	16384
Low Water Mark:	8192

The `pipemod` handles `M_FLUSH` messages in STREAMS/UX-based pipes. The `pipemod` is described in Chapter 4, “Modules and Drivers.”

HP-UX Changes to STREAMS/UX Data Structures

STREAMS/UX data structures are almost identical to those described in the *SVR4.2 Driver* manual. STREAMS/UX places additional restrictions on how some of these structures can be accessed. STREAMS/UX data structures that differ from the descriptions in the *SVR4.2 Driver* manual are described here. Data structures identical to those described in the *SVR4.2 Driver* manual are not listed here.

STREAMS/UX data structures contain some declarations for fields used by STREAMS/UX internally that are not visible to the STREAMS/UX programmer. The programmer will not be affected by these fields except that the `sizeof` function will return a larger value.

Message Data Structures

These structures are slightly different from the ones in the *SVR4.2 Driver* manual.

`msgb` defined in `<sys/stream.h>`

The `msgb` structure contains `MSG_KERNEL_FIELDS`, which defines fields used internally by STREAMS/UX.

`iocblk` defined in `<sys/stream.h>`

The `ioc_count` is defined to be a member of a union.

`copyreq` defined in `<sys/stream.h>`

The `cq_addr` is defined to be a member of a union.

`copyresp` defined in `<sys/stream.h>`

The `cp_rval` is defined to be a member of a union.

Queue Data Structure

The queue structure is slightly different from the one described in the *SVR4.2 Driver* manual. The structure is defined in the file `<sys/stream.h>`. It contains an additional field `QUEUE_KERNEL_FIELDS` which defines fields used internally by STREAMS/UX.The multiplexor ID number returned by `I_LINK` and `I_PLINK` is opaque to the user and not a small integer such as 0, 1, 2, 3.

STREAMS/UX Data Structure Restrictions

STREAMS/UX has the same restrictions as those described in the *Kernel Data Structure* chapter of the *SVR4.2 Driver* manual. Also, STREAMS/UX limits user written functions where users can access the queue structure directly. A queue's open, close, put, or service routine can manipulate the queue structure as specified by SVR4.2. On a uniprocessor system, a queue's entry points can access the other queue in the queue pair in the same way that they can access their own queue. On a multiprocessor system, a queue's entry points can manipulate queues belonging to entities with which they can share data. They can manipulate the queues in the same way that they can manipulate their own queue.

It is difficult to program other functions (besides those described) to access the queue structure directly, especially on multiprocessor systems. If a queue's entry points access queues other than those described previously, or if non-STREAMS/UX software processes data in a STREAMS/UX queue, use the `streams_put` utility to manipulate the queues safely. `streams_put` is described in the *"HP-UX Modifications to STREAMS / UX Utilities"* section of this appendix.

If `streams_put` cannot be used, the code that accesses a STREAMS/UX queue must, at a minimum, follow these additional rules:

- The software must ensure that it is accessing an allocated, opened queue.
- It cannot dereference the `q_first`, `q_last`, or `q_next` pointers. In other words, it cannot read or write data pointed at by the pointers. For example, the function can check if `q_first` is 0, but it cannot read the `q_first->b_next` field.
- Any additional synchronization required for the modules and drivers must be implemented to work correctly by the developer. The developers need to synchronize the function accessing the STREAMS/UX queue with the queue's entry points. This is because the function and the entry points may access the queue in parallel on a multiprocessor system and may interrupt each other while accessing the queue on a uniprocessor system.

HP-UX Changes to Message Types

STREAMS/UX supports all the message types as in SVR 4.2. However STREAMS/UX differs from SVR4.2 STREAMS only in the way it behaves on the receipt of `M_HANGUP` message at the stream head for controlling terminals. For the controlling terminals, the stream head sends a `SIGHUP` signal to the foreground process group and not the controlling terminal when it receives a `M_HANGUP` message.

In addition, STREAMS/UX also offers a the following message types:

Ordinary or Low Priority Message Types

M_TRAIL

This message is generated and sent upstream by a driver following the `M_HANGUP` message to mark the end of data after an `M_HANGUP` message. The `M_TRAIL` message will be processed at the stream head only if the preceding `M_HANGUP` message sent by the driver contained the 2-byte `TRAIL_TOKEN`.

High Priority Message Types

M_CLOSE

This message is generated by the stream head to notify the driver of `close(2)` when the driver specifies `C_ALLCLOSES` in the `d_flags` field of it's `drv_ops_t` structure. When the `C_ALLCLOSES` flag is set, STREAMS sends an M_CLOSE message downstream and waits for a reply from the driver. The stream will be dismantled only if the reply (`M_CLOSE_REPL`) for the `M_CLOSE` message has `MSGLASTCLOSE` set in the `b_flag` else `close(2)` returns without dismantling the stream.

M_CLOSE_REPL

This message is generated by the driver in response to the `M_CLOSE` message sent to it by the stream head. When the flag `C_ALLCLOSES` is set, it is the responsibility of driver to keep track of the information about the number of opens/closes for this device and accordingly set the `MSGLASTCLOSE` in `b_flag` in the message `M_CLOSE_REPL` if it wants the stream associated with device to be dismantled.

Changes to M_HANGUP

STREAMS/UX differs from SVR4.2 STREAMS in the way it behaves on the receipt of an `M_HANGUP` message at the stream head for controlling terminals. STREAMS/UX differs from SVR4.2 STREAMS in the way it behaves on the receipt of an M_HANGUP message at the stream head for controlling terminals. For the controlling terminals on receiving an M_HANGUP message at stream head, the stream head sends a SIGHUP signal to the foreground process group and not the controlling terminal.

If an `M_TRAIL` message type is used to mark the end of data after the `M_HANGUP` message, then the `M_HANGUP` message must contain the 2-byte `TRAIL_TOKEN`.

HP-UX Changes to Cloning

STREAMS/UX supports two methods of cloning. See the *SVR4.2 STREAMS* manual for more information about cloning. *Chapter 4, “Modules and Drivers,”* explains the two HP-UX cloning methods in detail.

The major number for the `clone` driver in HP-UX is 72 while SVR4.2 cloning uses 63 as the major number of the `clone` driver.

STREAMS/UX Hardware Driver Writing

STREAMS/UX does not provide all the kernel utilities needed to write a STREAMS hardware driver. STREAMS/UX provides only the utilities described in this manual. Customers who need to write STREAMS hardware drivers should refer to the *HP-UX Driver Development Guid*e for details.

Differences Between SVR4 MP and HP-UX MP STREAMS

HP-UX STREAMS provides MP scalability differently from SVR4 MP STREAMS. There are two main differences.

1. The first pertains to which STREAMS/UX entities run in parallel. SVR4 MP STREAMS executes put and service procedures for the same queue concurrently although only one instance of a service procedures can run at a time. HP-UX, unlike SVR4 MP, allows the developer to configure which STREAMS/UX entities run in parallel. The most parallelism that a STREAMS/UX developer can configure is to run entry points for different queues concurrently. Unlike SVR4 MP, HP-UX only allows one entry point for a queue to run at a time. The put and service procedures for the same queue cannot run in parallel. Also, multiple instances of a queue's put or service procedure cannot execute concurrently.
2. The second difference has to do with synchronizing access to module and driver private data structures. SVR4 MP STREAMS does not provide protection for private structures. The module or driver code uses spinlocks to synchronize access. STREAMS/UX provides protection for private structures. The developer configures the amount of concurrency for a module or driver based on the entities with which it shares data structures. For example, if all instances of a module access the same table, the programmer can configure the module so that only one instance runs at a time.

STREAMS/UX Synchronization Levels

HP-UX STREAMS supports MP scalable drivers and modules. The amount of synchronization for modules and drivers can be configured. Pick a level which is consistent with a module's or driver's use of shared data structures. STREAMS/UX provides six levels of parallelism which are called **queue**, **queue pair, module**, **elsewhere**, **global**, and **NoSync** (described in detail in *Appendix F, "Synchronization Levels."*)

The STREAMS/UX synchronization levels also apply to open and close. For example, if a module is configured for queue pair level synchronization, none of the put or service procedures for the queue pair can run at the same time as the queue pair's open or close. Also, open cannot run at the same time as close. The least amount of protection that STREAMS/UX provides for opens and closes is queue pair. Even if a module is configured with queue-level synchronization, it will run as if it were configured with queue pair-level synchronization during opens and closes.

STREAMS does not synchronize the running of `timeout` and `bufcall` callback functions with modules and drivers.

Synchronization levels can be used to protect module and driver private data structures as long as the driver/module is not configured at `SQLVL_NOSYNC`.

Strategies for Porting SVR4 MP Modules and Drivers to HP-UX

The best way to port SVR4 MP scalable modules and drivers to HP-UX is to change the SVR4 MP code to use the STREAMS/UX synchronization levels. First, analyze how the SVR4 MP code shares data structures, and then configure the modules and drivers to use synchronization levels which correctly serialize access to shared data. Defines can be used to change module and driver spinlock calls to no-ops. This approach is likely to get the best performance, but may require much effort. Also, the STREAMS/UX synchronization levels may not be suitable for all designs.

To make porting easier, STREAMS/UX will provide support for the SVR4 MP spinlock primitives. SVR4 MP modules and drivers could be ported to HP-UX by configuring them to run with queue synchronization and leaving in the calls to SVR4 MP spinlock routines. A disadvantage of this porting strategy is that it may not achieve as much performance as the first. Some of the synchronization provided by STREAMS/UX will be redundant with the synchronization implemented by module and driver spinlocks. In some cases, a combination of these two strategies may make sense. For example, suppose several modules and drivers share the same structure, but do not access it on the main read and write paths. Use SVR4 MP spinlocks to protect this data, but use the STREAMS/UX synchronization levels to protect other structures.

The STREAMS/UX Scheduler

The STREAMS/UX scheduler runs service routines that are scheduled by STREAMS/UX utilities such as `putq`. The scheduler will run all scheduled service procedures before returning to user level. The scheduler is a real time daemon that runs at priority 100. (A low priority number denotes a high priority. For example, a priority number of 50 would be of higher priority than the number 100). STREAMS/UX applications need to run at a lower priority (higher priority number) than the STREAMS/UX scheduler; otherwise service procedures will not run before the scheduler returns to user level from the kernel.

F Synchronization Levels

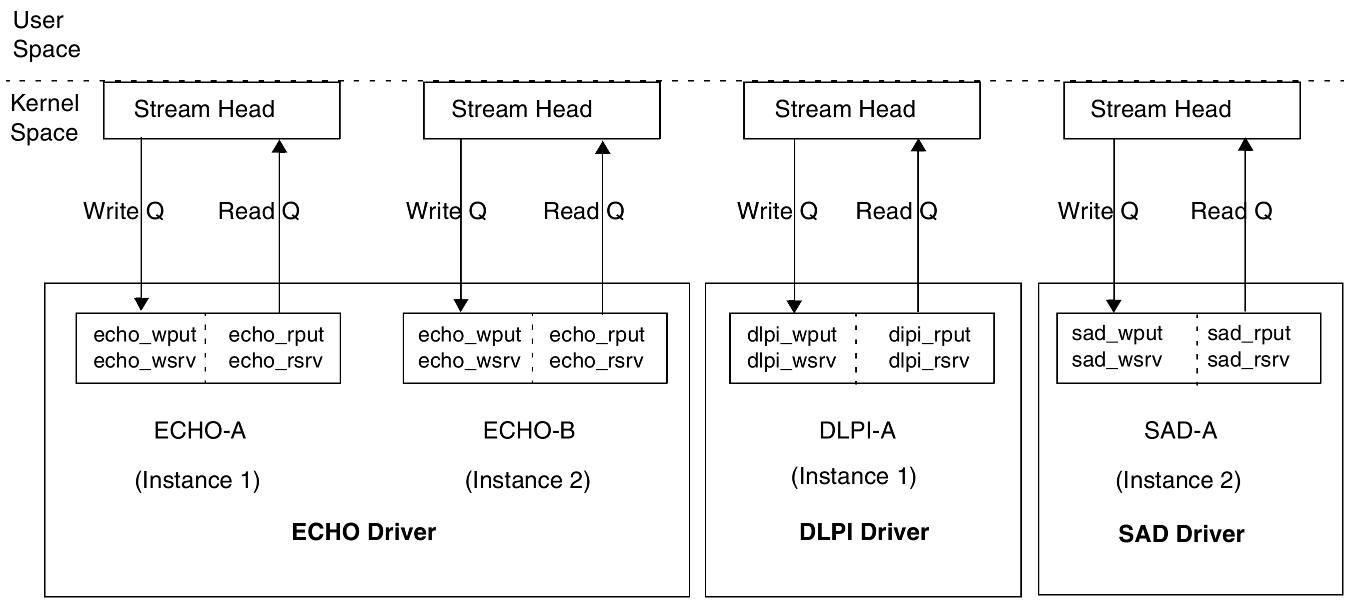
STREAMS/UX supports MP scalable drivers and modules. Modules and drivers need to select the level of parallelism consistent with their use of shared data structures. STREAMS/UX allows the degree of parallelism to be configured by providing the following levels of synchronization:

1. Queue level synchronization — `SQLVL_QUEUE`
2. Queue-pair level synchronization — `SQLVL_QUEUEPAIR`
3. Module level synchronization — `SQLVL_MODULE`
4. Elsewhere level synchronization — `SQLVL_ELSEWHERE`
5. Global level synchronization — `SQLVL_GLOBAL`
6. NoSync level synchronization — `SQLVL_NOSYNC`

HP-UX requires all modules and drivers to specify their synchronization level in the `inst_sync_level` field in the `streams_info_t` structure.

Figure F-1, "Synchronization Levels," shows the different synchronization levels.

Figure F-1 Synchronization Levels



Queue Level Synchronization

The queue level synchronization serializes access to a queue so that only one request is processed at a time. Requests to different queues can run in parallel.

For example, if the echo driver in *Figure F-1, "Synchronization Levels,"* is configured for queue synchronization, the following procedures will not execute concurrently:

- ECHO-A's `echo_rput` and `echo_rsrv` procedures.
- ECHO-A's `echo_wput` and `echo_wsrv` procedures.

However, in this synchronization mode, STREAMS/UX will run the following concurrently:

- ECHO-A's `echo_rput` and ECHO-A's `echo_wput` procedures.
- Any of ECHO-A's procedures with any of ECHO-B, DLPI-A or SAD-A's procedures.

If a module uses queue synchronization, a queue's put and service procedures can share data with each other. This occurs because STREAMS/UX does not execute these procedures concurrently.

Queue-Pair Level Synchronization

The queue-pair level synchronization serializes the read-and write-queue pair of a module/driver. This level of synchronization ensures that only one request to this queue pair is processed at a time. Requests to different queue pairs can run in parallel.

For example, if the echo driver in *Figure F-1, "Synchronization Levels,"* is configured for queue-pair synchronization, then at any given time, only one of ECHO-A's `echo_rput`, `echo_rsrv`, `echo_wput`, and `echo_wsrv` can run. However, an ECHO-A routine can run in parallel with an ECHO-B routine.

If a module uses queue-pair synchronization, the read queue and write queue can share data with each other.

Module Level Synchronization

The module level synchronization serializes access to all the instances of the module. This synchronization ensures that only one request for the instances of this module can be processed at a time. Requests to different modules can run in parallel.

For example, in *Figure F-1, "Synchronization Levels,"* if the echo driver is configured for module synchronization, then only one request to any ECHO-A or ECHO-B will be processed at a time. However, STREAMS/UX will be able to process the requests to a dlpi or sad driver at the same time.

With module synchronization, all instances of this module can share data with each other.

Elsewhere Level Synchronization

The elsewhere level synchronization serializes a group of different modules. This synchronization ensures that only one request to the entire group of modules can be processed at a time. Requests to other queues that are not in this group can run concurrently.

For example, if the echo and dlpi drivers in *Figure F-1, "Synchronization Levels,"* are configured to be members of an elsewhere synchronization group, and the sad driver is configured to be in a different elsewhere group, then only one driver function in ECHO-A, ECHO-B and DLPI-A can run at a time. However, a function in ECHO-A, ECHO-B or DLPI-A can run in parallel with a function in SAD-A.

Modules in a group can share data, since no two functions in a module group can run concurrently under the elsewhere level synchronization.

Global Level Synchronization

The global level synchronization serializes all the requests to the modules configured at the global level synchronization. Only one request to all modules under the global level synchronization can be processed at a time. Requests to modules that are not configured under global level synchronization can run concurrently.

For example, in *Figure F-1, "Synchronization Levels,"* if the echo, dlpi, and sad drivers are configured for global synchronization, only one driver function in ECHO-A, ECHO-B, DLPI-A, and SAD-A can run at a time. However, one of these drivers could run in parallel with a module configured for a different synchronization level.

All modules configured with global synchronization can share data.

Global level synchronization may cause a system-wide performance impact. Therefore, it is highly recommended not to use global level synchronization.

Nosync level synchronization

Nosync level synchronization provides the highest level of concurrency. For a given queue, it allows multiple requests to be processed at the same time. Requests to different queues can run in parallel.

For example, if the echo driver in Figure F-1 is configured for nosync synchronization, STREAMS/UX cannot run the following procedures concurrently:

- Multiple instances of ECHO-A's echo_rsrv procedures.
- Multiple instances of ECHO-A's echo_wsrv procedures.

However, in this synchronization mode, STREAMS/UX can run the following procedures concurrently:

- Multiple instances of ECHO-A's echo_rput procedures.
- Multiple instances of ECHO-A's echo_wput procedures
- Multiple instances of ECHO-A's echo_rput and a single instance of echo_rsrv procedures.

- Multiple instances of ECHO-A's echo_wput and a single instance of echo_wsrv procedures.
- Any of ECHO-A's procedures with any of ECHO-B, DLPI-A, or SAD-A's procedures.

If a module uses nosync synchronization, STREAMS/UX can concurrently execute multiple instances of a queue's put procedure and a single instance of the same queue's service procedure. This necessitates that the modules protect the module-specific data that has to be shared between multiple instances of the put procedures, or between the put and the service procedures.

STREAMS/UX does not guarantee the order of messages coming out of the modules/drivers using nosync synchronization, because it has no control of the execution order when multiple instances of the put and service procedures are executed simultaneously. If the order of the messages matter, than the modules/drivers should device their own mechanisms to achieve it.

STREAMS/UX in general will maintain the integrity of its private data structures. However when certain operations manipulating the queue directly namely insq(), rmvq(), strqset() and strqget() are to be used in a module/driver using nosync synchronization, an explicit call to freezestr() is necessary to maintain the integrity of the STREAMS data structures.

Refer to Appendix G, “STREAMS Commands,” for more details on these utilities.

G STREAMS Commands

Overview

This appendix discusses the various STREAMS/UX commands.

autopush

The autopush command manages the system database of automatically pushed STREAMS modules. The HP-UX autopush command has been enhanced with respect to UNIX SVR4.2, to allow the user to specify the device name in place of the major number, which is recommended since HP-UX provides dynamic major numbers. The name can be specified in the autopush file and on the command line. Device names are located in the HP-UX modmeta files. The major number can still be used if needed.

Synopsis

The general syntax of `autopush` on HP-UX is as follows:

```
autopush   -f autopush_file_name

autopush   -r   -M   major_num|dev_name   -m   minor_num

autopush   -g   -M   major_num|dev_name   -m   minor_num
```

The contents of the `autopush` filename are as follows:

```
major_num|dev_name low_minor high_minor mod_name 1...mod_name N
```

Options and Arguments

`autopush` recognizes the following command-line options and arguments.

`-f file` Using the configuration information contained in file, load the system database with the names of the STREAMS devices and a list of modules to use for each device. When a device is subsequently opened, the HP-UX STREAMS subsystem pushes the modules onto the stream for the device.

If “-” appears as a file argument, autopush uses the standard input.

file must contain one or more lines of at least four fields separated by a space as shown below:

`major minor lastminor module1 module2 ... moduleN` The first field major can be either an integer or a device name. The device name is the name for the device used in the modmeta file. The next two fields are integers. If `minor` is set to -1, then all minor devices for the specified major are configured and `lastminor` is ignored. If `lastminor` is 0, then only a single `minor` device is configured.

To configure a range of minor devices for a `major` device, `minor` must be less than `lastminor`. The remaining field(s) list one or more module names. Each module is pushed in the order specified. A maximum of eight modules can be pushed. Any text after a “#” character in file is treated as a comment for that line only.

This option is also used to restore device configuration information previously removed by autopush -r. However, when used in such a manner, the entire database is restored, not just the information that was previously removed.

`-g -M major -m minor` Display current configuration information from the system database for the STREAMS device specified by the major device number (or device name for the device from the `modmeta` file) and minor number.

If a range of minors has been previously configured then `autopush -g` returns the configuration information for the first minor in the range, in addition to other information.

`-r -M major -m minor` Remove configuration information from the system database for the STREAMS device specified by the major device number (or device name for the device from the `modmeta` file and `minor` number). Removal is performed on the database only, not on the original configuration file. Therefore, the original configuration can be restored by using the `-f` file option. To permanently exclude a STREAMS device from the database, its information must be removed from the configuration file.

If `minor` matches the first minor of a previously configured range then `autopush -r` removes the configuration information for the entire configured range.

Examples

The file `/tmp/autopush.example` contains the following:

```
75 -1 0 modA

modB test 0 5 modC modA
```

`autopush -f /tmp/autopush.example` will cause `modA` and `modB` to be pushed whenever major device `# 75` is opened, and `modC` and `modA` to be pushed for the first six opens of `test`.

The following example lists information about the stream for major device `75` and its minor device `2`:

```
autopush -g -M 75 -m 2
```

strace and strerr

The `strace` and `strerr` commands use the STREAMS log driver, `/dev/strlog`. SVR4.2 calls this driver `/dev/log`, but HP-UX already includes a non-streams driver named `/dev/log`. Therefore, STREAMS logging uses `/dev/strlog`.

strace

strace gets STREAMS event trace messages from STREAMS drivers and modules via the STREAMS log driver (`strlog(7)`), and writes these messages to standard output. By default, `strace` without arguments writes all STREAMS `trace` messages from all drivers and modules. `strace` with command-line arguments limits the `trace` messages received.

The syntax for the strace command is as follows:

```
strace [ mod sub pri ] ...
```

The following arguments must be specified in groups of three:

`mod`	Specifies the STREAMS module identification number from the `streamtab` entry.
`sub`	Specifies a sub-identification number that often corresponds to a minor device.
`pri`	Specifies a tracing priority level. `strace` gets messages of a level equal to or less than the value specified by `pri`. Only positive integer values are allowed.

The value "`all`" can be used for any argument in the `strace` command line to indicate that there are no restrictions for that argument.

Multiple sets of the three arguments can be specified to obtain the messages from more than one driver or module.

Only one `strace` process can open the STREAMS log driver at a time. When `strace` is invoked, the log driver compares the sets of command line arguments with actual `trace` messages. The log driver returns only the messages that satisfy the specified criteria.

STREAMS event `trace` messages have the following syntax:

```
seq time tick pri ind mod sub text
```

The components are interpreted as follows:

`seq`	`trace` event sequence number.
`time`	Time in hh:mm:ss when the message was sent
`tick`	Time when the message was sent, expressed in terms of machine ticks since the last boot.
`pri`	Tracing priority level as defined by the STREAMS driver or module that created the messages.
`ind`	Can be any combination of the following three message indicators: `E`: The message has also been saved in the error log. `F`: The message signalled a fatal error. `N`: The message has also been mailed to the system administrator.
`mod`	Module identification number of the `trace` message source.
`text`	`trace` message text.

`strace` runs until it is terminated by the user.

Example

The following examples displays all `trace` messages from the driver or module identified by `mod 28`:

```
strace 28 all all
```

Example

The following example displays `trace` messages of any tracing priority level from the driver or module identified by mod 28, and its minor devices identified by the sub 2, 3, or 4:

```
strace  28 2 all  28 3 all  28 4 all
```

Example

The following example displays the trace messages from the same driver or module and subs. This example limits the priority levels as follows:to 0 for subs 2 and 3; 1 for sub 4, driver or module 28:

- 0 for subs 2 and 3
- 1 for subs 4, driver, or mod 28

```
strace  28 2 0  28 3 0  28 4 1
```

NOTE Running `strace` with several sets of arguments can impair STREAMS performance, particularly for those modules and drivers that are sending the messages. `strace` may not be able to handle a large number of messages. Some of the messages may be lost if drivers and modules return messages to `strace` too quickly.

strerr

`strerr` receives error messages from the STREAMS log driver (`strlog(7)`). These error messages are added to the STREAMS error log files (`error.mm-dd`) in the STREAMS error logger directory (`/var/adm/streams` by default). On the first call to `strerr`, it creates the log file `error.mm-dd`. This is a daily log file, where `mm` indicates the month and `dd` indicates the day of the logged messages. `strerr` then appends the error messages to the log file as they are received from the STREAMS log driver.

The syntax for the strerr command is as follows:

```
strerr [-a sys_admin_mail_name] [-d logdir]
```

`-a sys_admin_mail_name`	Specifies tthe user's mail name for sending mail messages. Mail is sent to the system administrator by default.
`-d logdir`	Specify the directory to contain the error log file. The default directory is `/var/adm/streams`.

STREAMS error log messages have the following syntax:

```
seq time tick pri ind mod sub text
```

The components are interpreted as follows:

`seq`	`trace` event sequence number.
`time`	Time in hh:mm:ss when the message was sent
`tick`	Time when the message was sent, expressed in terms of machine ticks since the last boot.

`pri`	Tracing priority level as defined by the STREAMS driver or module that created the messages.
`ind`	Can be any combination of the following three message indicators: `T`: The message has also been saved in the trace log. `F`: The message signalled a fatal error. `N`: The message has also been mailed to the system administrator.
`mod`	Module identification number of the `trace` message source.
`text`	`trace` message text.

`strace` runs until it is terminated by the user.

NOTE `strerr` runs continuously until terminated by the user. Only one `strerr` process at a time can open the STREAMS log driver. This restriction is intended to maximize performance. The STREAMS error logging mechanism works best when it is not overused. `strerr` can degrade STREAMS performance by affecting the response, throughput, and other behaviors of the drivers and modules that invoke it. `strerr` also fails to capture messages if drivers and modules generate messages at a higher rate than its optimum read rate. If there are missing sequence numbers among the messages in a log file, messages have been lost.

strchg and strconf

The strchg and strconf commands are used to change or query the configuration of the stream associated with the user's standard input. The strchg command pushes modules on and/or pops modules off the stream. The strconf command queries the configuration of the stream. Only a user with appropriate privileges or owner of a STREAMS device may alter the configuration of that stream.

The syntax options for the `strchg` command are:

```
strchg -h module1 [, module2]...

strchg -p [ -a|-u module]

strchg -f file
```

-h module1[,module2] ... strchg pushes modules onto a stream. The modules are pushable STREAMS modules as defined by module1, module2, and so on. The modules are pushed in order. That is, module1 is pushed first, module2 is pushed second, etc.

`-p` `strchg` pops the topmost module from the stream.

`-p -a` `strchg` pops all the modules above the topmost driver.

`-p -a` `strchg` pops all the modules above the topmost driver.

`-p -u` `strchg` pops all the modules above `module`.

The `-a` and `-u` options are mutually exclusive.

`-f file` The user can specify a file that contains a list of modules representing the desired configuration of the stream. Each module name must appear on a separate line where the first name represents the topmost module and the last name represents the module that should be closest to the driver. The strchg command will determine the current configuration of the stream and pop and push the necessary modules in order to end up with the desired configuration.

The -`h`, `-f`, and `-p` options are mutually exclusive.

NOTE If the user is neither the owner of the stream nor a user with appropriate privileges, the `strchg` command will fail. If the user does not have read permissions on the stream or appropriate privileges, the `strconf` command will fail. If modules are pushed in the wrong order, the stream may not function as expected. For `ttys`, if the line discipline module is not pushed in the correct place, the terminal may not respond to any command.

Diagnostics

`strchg` returns zero on success. It prints an error message and returns non-zero status for various error conditions, including usage error, bad module name, too many modules to push, failure of an ioctl on the stream, or failure to open file from the `-f` option.

`strconf` returns zero on success (for the `-m` or `-t` option, success indicates that the named or topmost module is present).

`strconf` returns a non-zero status if invoked with the `-m` or `-t` option, and if the module is not present.

`strconf` prints an error message and returns non-zero status for various error conditions, including usage error or failure of an ioctl on the stream.

```
75 -1 0 modA
```

```
modB test 0 5 modC modA
```

`autopush -f /tmp/autopush.example` will cause `modA` and `modB` to be pushed whenever major device `# 75` is opened, and `modC` and `modA` to be pushed for the first six opens of `test`.

The following example lists information about the stream for major device `75` and its minor device `2`:

```
autopush -g -M 75 -m 2
```

Examples

The following command pushes the module ldterm on the stream associated with the user's standard input:

```
strchg -h ldterm
```

The following command pops the topmost module from the stream associated with `/dev/term/24`.

```
strchg -p < /dev/term/24
```

The user must be the owner of this device or be a user with appropriate privileges.

If `fileconf` contains `compat`, `1dterm`, and `pterm` use the following command:

```
strchg -f fileconf
```

The user’s standard input stream will be configured so that the module `pterm` is pushed over the driver. This is followed by `1dterm` and `compat` closest to the stream head.

The `strconf` command with no arguments lists the modules and topmost driver on the stream. For a stream that only has the module `1dterm` pushed above the `ports` driver, it would produce the following output.

```
1dterm
ports
```

The following command asks if `ldterm` is on the stream:

```
strconf -m ldterm
```

The following output is produced while returning an exit status of `0: yes`

strclean

`strclean` cleans the STREAMS error logger directory of log files (`error.mm-dd`) that contain error messages sent by the STREAMS log driver `strlog(7)`. If the `-d` option is not used to specify another directory, `strclean` removes the error log files in the `/var/adm/streams` directory. If the `-a` option is not used to specify another age, `strclean` removes the error log files that have not been modified over the last three days.

The syntax for the `strclean` command is as follows:

```
strclean [-d logdir] [-a age]
```

strclean recognizes the following options and command-line arguments:

`-d logdir`	Specifies a directory for the location of the STREAMS error log files to be removed, if the default directory `/var/adm/streams` is not specified.
`-a age`	Specifies a maximum age in days for the STREAMS error log files, if the default age of 3 is not specified. The value of `age` must be an integer greater than or less than 3.

Example

The following command will remove error messages that are one day old, from the `/tmp/streams` directory:

```
strclean -d /tmp/streams -a 1
```

H STREAMS Kernel Tunable Parameters

Overview

The following kernel parameters can be configured and used for managing the resources used by STREAMS.

`NSTREVENT`	Maximum number of outstanding streams bufcalls that are allowed to exist at any given time. This number should be greater than or equal to the maximum number of bufcalls that can be generated by all modules pushed onto a given stream. This serves to limit runaway bufcalls.
`NSTRPUSH`	Maximum number of streams modules that are allowed to exist in any single stream at any given time. This provides a mechanism for preventing a software defect from attempting to push too many modules onto a stream. It does not protect against the malicious use of streams.
`NSTRSCHED`	Maximum number of streams scheduler daemons that are allowed to run at any given time. This value is related to the number of processors installed in the system.
`STRCTLSZ`	Maximum number of control bytes allowed in the control portion of any streams message.
`STRMSGSZ`	Maximum number of bytes that can be placed in the data portion of any streams message.
`streampipes`	Maximum number of bytes that can be placed in the data portion of any streams message.

These kernel tunable parameters can be either modified with SAM or by using the `kctune(1M)` command.

The details for each of these kernel tunable parameters are provided in the following manpages.

NSTREVENT

The maximum number of outstanding STREAMS bufcalls.

Values

Failsafe

50

Default

50

Allowed

0 - 2147483647

Recommended

50

Description

This tunable limits the maximum number of outstanding bufcalls that are allowed to exist at any given time. This tunable is intended to protect the system against resource overload caused by the combination of modules running in all streams issuing an excessive number of bufcalls. The value selected should be equal to or greater than the combined maximum number of bufcalls that can be reasonably expected during normal operation from all streams. bufcalls are used by STREAMS modules in low memory situations.

Who is Expected to Change This Tunable?

Any customer.

Restrictions on Changing

Changes to this tunable take effect at the next reboot.

When Should the Value of This Tunable Be Raised?

When the customer needs to push more STREAMS modules in a single stream.

What are the Side Effects of Raising the Value of This Tunable?

Runaway applications may unduly consume system resources.

When Should the Value of This Tunable be Lowered?

There is no reason to lower the value below the default value.

What are the Side Effects of Lowering the Value of This Tunable?

Network commands may fail if the value is too low.

What Other Tunable Should be Changed at the Same Time?

None.

Warnings

All HP-UX kernel tunable parameters are release specific. This parameter may be removed or have its meaning changed in future releases of HP-UX.

Author

`NSTREVENT` was developed by HP.

NSTRPUSH

The maximum number of STREAMS modules in a single stream.

Values

Failsafe

16

Default

16

Allowed

0 - 2147483647

Recommended

16

Description

This tunable defines the maximum number of STREAMS modules that can be pushed onto a stream. This provides some protection against run-away processes that might automatically select modules to push onto a stream. It is not intended as defense against malicious use of STREAMS modules by system users.

Most systems do not require more than about three or four modules in a stream. However, there may be some unusual cases where more modules are needed. The default value for this tunable allows as many as 16 modules in a stream, which should be sufficient for even the most demanding installations and applications.

Who is Expected to Change This Tunable?

Any customer.

Restrictions on Changing

Changes to this tunable take effect at the next reboot.

When Should the Value of This Tunable Be Raised?

When the system experiences a lot of low memory situations.

What are the Side Effects of Raising the Value of This Tunable?

If too big a number is chosen, the STREAMS subsystem preallocates more memory for internal data structures than necessary. This reduces the amount of memory available to applications and the system.

When Should the Value of This Tunable be Lowered?

If the tunable is increased for a particular STREAMS module/driver, this tunable can be lowered when that STREAMS module/driver is removed. It should be returned to its previous value. However, HP does not recommend a value lower than the default value.

What are the Side Effects of Lowering the Value of This Tunable?

System performance will be lowered during low memory situations.

What Other Tunable Should be Changed at the Same Time?

None.

Warnings

All HP-UX kernel tunable parameters are release specific. This parameter may be removed or have its meaning changed in future releases of HP-UX.

Author

`NSTRPUSH` was developed by HP.

NSTRSCHED

The number of STREAMS scheduler daemons to be run.

Values

Failsafe

0

Default

0

Allowed

0 - 2147483647

Recommended

0

Description

This tunable defines the number of STREAMS scheduler daemons to be run on a system.

If the tunable value is set to zero, the system determines the number of daemons to run based on the number of processors in the system. A positive non zero tunable value is the number of "smpsched" daemons that will be created on an MP system.

NOTE This tunable is for use by specific HP products only. It may be removed in future HP-UX releases.

Who is Expected to Change This Tunable?

Any customer.

Restrictions on Changing

Changes to this tunable take effect at the next reboot.

When Should the Value of This Tunable Be Raised?

This tunable is for use by specific HP products only. It may be removed in future HP-UX releases. *Is there any specific reason why more STREAMS scheduler daemons should be run?*

What are the Side Effects of Raising the Value of This Tunable?

It could change the system performance unpredictably.

When Should the Value of This Tunable be Lowered?

This tunable is for use by specific HP products only. It may be removed in future HP-UX releases.

What are the Side Effects of Lowering the Value of This Tunable?

It could change the system performance unpredictably.

What Other Tunable Should be Changed at the Same Time?

None.

Warnings

This tunable is for use by specific HP products only.

All HP-UX kernel tunable parameters are release specific. This parameter may be removed or have its meaning changed in future releases of HP-UX.

Author

`NSTRSCHED` was developed by HP.

STRCTLSZ

The maximum size of streams message control (bytes).

Values

Failsafe

1024

Default

1024

Allowed

0 - 2147483647

Recommended

1024

Description

`STRCTLSZ` limits the maximum number of bytes of control data that can be inserted by `putmsg()` in the control portion of any streams message on the system. If the tunable is set to zero, there is no limit on how many bytes can be placed in the control segment of the message.

`putmsg()` returns `ERANGE` if the buffer being sent is larger than the current value of `STRCTLSZ`.

Who is Expected to Change This Tunable?

Any customer.

Restrictions on Changing

Changes to this tunable take effect at the next reboot.

When Should the Value of This Tunable Be Raised?

The tunable should be increased if the STREAMS modules or drivers require more bytes than the current value in the control portion of any streams message.

What are the Side Effects of Raising the Value of This Tunable?

The kernel will use more memory. During low memory situations, it may bring reduce system performance due to frequent swapping.

When Should the Value of This Tunable be Lowered?

The tunable could be lowered if the STREAMS modules or drivers do not require a longer message size than the current value in the control portion.

What are the Side Effects of Lowering the Value of This Tunable?

Improper functioning in any of the STREAMS modules or drivers may result. There may be some performance degradation, particularly in networking.

What Other Tunable Should be Changed at the Same Time?

None.

Warnings

This tunable is for use by specific HP products only.

All HP-UX kernel tunable parameters are release specific. This parameter may be removed or have its meaning changed in future releases of HP-UX.

Author

`STRCTLSZ` was developed by HP.

STRMSGSZ

The maximum size of streams message data (bytes).

Values

Failsafe

0

Default

0

Allowed

0 - 2147483647

Recommended

0

Description

This tunable limits the number of bytes of message data that can be inserted by putmsg() or write() in the data portion of any streams message on the system. If the tunable is set to zero, there is no limit on how many bytes can be placed in the data segment of the message.

`putmsg()` returns `ERANGE` if the buffer being sent is larger than the current value of `STRCTLSZ`. `write()` segments the data into multiple messages.

Who is Expected to Change This Tunable?

Any customer.

Restrictions on Changing

Changes to this tunable take effect at the next reboot.

When Should the Value of This Tunable Be Raised?

The tunable should be increased if the STREAMS modules or drivers require more bytes than the current value in the data portion of any streams message.

What are the Side Effects of Raising the Value of This Tunable?

The kernel will use more memory. During low memory situations, it may bring reduce system performance due to frequent swapping.

When Should the Value of This Tunable be Lowered?

The tunable could be lowered if the STREAMS modules or drivers do not require a longer message size than the current value in the data portion.

What are the Side Effects of Lowering the Value of This Tunable?

Improper functioning in any of the STREAMS modules or drivers may result. There may be some performance degradation, particularly in networking.

What Other Tunable Should be Changed at the Same Time?

None.

Warnings

This tunable is for use by specific HP products only.

All HP-UX kernel tunable parameters are release specific. This parameter may be removed or have its meaning changed in future releases of HP-UX.

Author

`STRMSGSZ` was developed by HP.

streampipes

Forces all pipes to be STREAMS-based.

Values

Failsafe

0

Default

0

Allowed

0 - 2147483647

Recommended

0

Description

This tunable determines the type of pipe that is created by the `pipe()` system call. If set to the default value of zero, all pipes created by `pipe()` are normal HP-UX file-system pipes. If the value is non-zero, `pipe()` creates STREAMS-based pipes, and STREAMS modules can be pushed onto the resulting stream.

If this tunable is set to a non-zero value, the `pipemod` and `pipedev` module and driver must be configured in `/stand/system`.

Who is Expected to Change This Tunable?

Any customer.

Restrictions on Changing

Changes to this tunable take effect at the next reboot.

When Should This Tunable be Switched On?

If the customer uses applications that require STREAMS-based pipes, this tunable should be switched on.

What are the Side Effects of Switching On This Tunable?

STREAMS-based pipes performance may differ from normal file system pipes.

When Should This Tunable be Switched Off?

If the customer does not need the STREAMS-based pipes, this tunable should be turned off.

What are the Side Effects of Switching Off This Tunable?

Applications that try to push STREAMS modules onto the pipe will fail.

What Other Tunable Should be Changed at the Same Time?

If this tunable is set to a non-zero value, the `pipemod` and `pipedev` module and driver must be configured in `/stand/system`.

Warnings

This tunable is for use by specific HP products only.

All HP-UX kernel tunable parameters are release specific. This parameter may be removed or have its meaning changed in future releases of HP-UX.

Author

`streampipes` was developed by HP.