STREAMS Mechanism 3

STREAMS Mechanism Overview

This chapter shows how to construct, use, and dismantle a Stream using STREAMS-related systems calls. General and STREAMS-specific system calls provide the user-level facilities required to make application programs. This system-call interface is upwardly compatible with the traditional character I/O facilities. The `open`(2) system call recognizes a STREAMS file and creates a Stream to the specified driver. A user process can receive and send data on STREAMS files using `read`(2) and `write`(2) in the same manner as with traditional character files. The `ioctl`(2) system call enables users to perform functions specific to a particular device. STREAMS `ioctl` commands (see `streamio`(7)) support a variety of functions for accessing and controlling Streams. The final `close`(2) on a Stream dismantles it.

In addition to the traditional `ioctl` commands and system calls, there are other system calls used by STREAMS. The `poll`(2) system call provides users with a mechanism for multiplexing input/output over a set of file descriptors that reference open files. The `putmsg`(2) and `getmsg`(2) and the `getpmsg`(2) and `putpmsg`(2) system calls enable users to send and receive STREAMS messages, and are suitable for interacting with STREAMS modules and drivers through a service interface.

STREAMS provides kernel facilities and utilities to support development of modules and drivers. The Stream head handles most system calls so that the related processing does not have to be incorporated in a module or driver.

3

Note – For the complete list of manual pages, please refer to Appendix F, “Manual Pages". Sections 1, 2, 3, 7, and 9 contain all the STREAMS information.

STREAMS System Calls

The STREAMS-related system calls are:

`open`(2)	Open a Stream
`close`(2)	Close a Stream
`read`(2)	Read data from a Stream
`write`(2)	Write data to a Stream
`ioctl`(2)	Control a Stream
`getmsg`(2)	Receive a message at the Stream head
`getpmsg`(2)	Receive a priority message at the Stream head
`putmsg`(2)	Send a message downstream
`putpmsg`(2)	Send a priority message downstream
`poll`(2)	Identify files on which a user can send or receive messages, or on which certain events have occurred (not restricted to streams, although historically was)
`pipe`(2)	Create a bidirectional channel that provides a communication path between multiple processes

Stream Construction

STREAMS constructs a Stream as a linked list of kernel-resident data structures. The list is created as a set of linked queue pairs. The first queue pair is the head of the Stream and the second queue pair is the end of the Stream. The end of the Stream represents a device driver, pseudo device driver, or the other end of a STREAMS-based pipe. Kernel routines interface with the Stream head to perform operations on the Stream. Figure 3-1 shows the upstream (read) and downstream (write) portions of the Stream. Queue H2 is the

upstream half of the Stream head and queue H1 is the downstream half of the Stream head. Queue E2 is the upstream half of the Stream end and queue E1 is the downstream half of the Stream end.

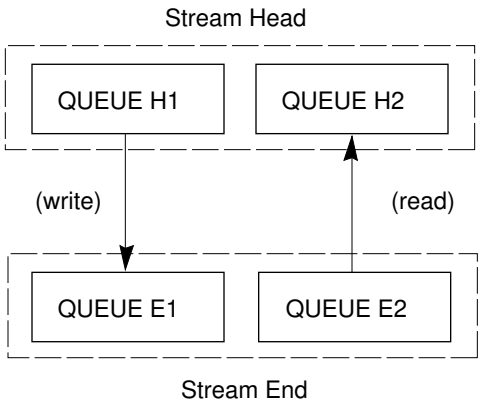

Figure 3-1 Upstream and Downstream Stream Construction

At the same relative location in each queue is the address of the entry point, a procedure to process any message received by that queue. The procedure for queues H1 and H2 processes messages sent to the Stream head. The procedure for queues E1 and E2, processes messages received by the other end of the Stream, the Stream end (tail). Messages move from one end to the other, from one queue to the next linked queue, as the procedure specified by that queue is executed.

Figure 3-2 shows the data structures forming each queue: `queue`(9S), `qinit`, `qband`, `module_info`, and `module_stat`. The `qband` structures have information for each priority band in the queue. The `queue` data structure contains various modifiable values for that queue. The `qinit` structure contains a pointer to the processing procedures, the `module_info` structure contains initial limit values, and the `module_stat` structure is used for statistics gathering. Each queue in the queue pair contains a different set of these data structures. There is a `queue`, `qinit`, `module_info`, and `module_stat` data structure for the upstream portion of the queue pair and a set of data structures for the downstream portion of the pair. In some situations, a queue pair may share some or all of the data structures. For

example, there may be a separate `qinit` structure for each queue in the pair and one `module_stat` structure that represents both queues in the pair. These data structures are described in Appendix A, “STREAMS Data Structures".

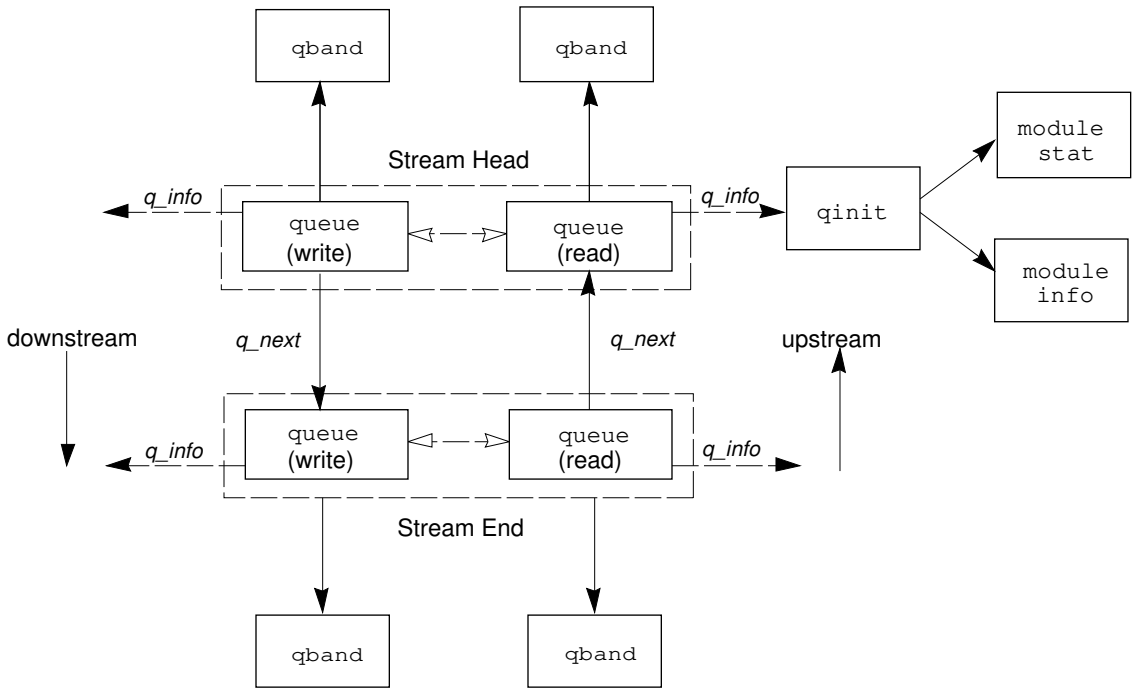

Figure 3-2 Stream Queue Relationship

Figure 3-2 shows two neighboring queue pairs with links (solid vertical arrows) in both directions. When a module is pushed onto a Stream, STREAMS creates a queue pair and links each queue in the pair to its neighboring queue in the upstream and downstream direction. The linkage allows each queue to locate its next neighbor. This relation is implemented between adjacent queue pairs by the *q_next* pointer. Within a queue pair, each `queue` locates its mate (see dashed arrows in Figure 3-2) by use of STREAMS functions, since there is no pointer between the two `queues`. The existence of the Stream head and Stream end is known to the queue procedures only as destinations towards which messages are sent.

Opening a STREAMS Device File

One way to construct a Stream is to open (see open(2)) a STREAMS-based special file. All entry points into the driver are defined by the streamtab structure (streamtab(9S)) for that driver. The streamtab structure is listed here.

```
struct streamtab {
    struct qinit           *st_rdinit;  /* read QUEUE */
    struct qinit           *st_wrinit;  /* write QUEUE */
    struct qinit           *st_muxrinit;/* lower read QUEUE */
    struct qinit           *st_muxwinit;/* lower write QUEUE*/
};
```

The streamtab structure defines a module or driver. *st_rdinit* points to the read qinit structure for the driver and *st_wrinit* points to the driver's write qinit structure. *st_muxrinit* and *st_muxwinit* point to the lower read and write qinit structures if the driver is a multiplexer driver.

If the open call is the initial file open, a Stream is created. (There is one Stream per major/minor device pair.)

A Stream head is created from a data structure and a pair of queue structures. The content of stdata and queue are initialized with predetermined values, including the Stream head processing procedures.

There is one Stream head per Stream. The Stream head is used by STREAMS while performing operations on the Stream.

A queue structure pair is allocated for the Stream head. The queue limits are initialized to those values specified in the corresponding module_info structure. The queue processing routines are initialized to those specified by the corresponding qinit structure.

Then, the *q_next* values are set so that the Stream head write queue points to the driver write queue and the driver read queue points to the Stream head read queue. The *q_next* values at the ends of the Stream are set to null. Finally, the driver open procedure (located via its read qinit structure) is called.

If this open is not the initial open of this Stream, the only actions performed are to call the driver open and the open procedures of all pushable modules on the Stream. When a Stream is already open, further opens of the same device will result in the open routines of all modules and the driver on the Stream

being called. Note that this is in reverse order from the way a Stream is initially set up. That is, a driver is opened and a module is pushed on a Stream. When a push occurs the module open routine is called. If another open of the same device is made, the open routine of the module will be called followed by the open routine of the driver. This is opposite from the initial order of opens when the Stream is created.

A new feature of STREAMS is `autopush`. Upon an `open(2)` system call, a preconfigured list is checked for modules to be pushed. All modules in this list are pushed before the `open()` returns. For more information, see `autopush(1M)` and `sad(7)`.

Creating a STREAMS-based Pipe

In addition to opening a STREAMS-based driver, a Stream can be created by creating a pipe (see `pipe`(2)). Since pipes are not character devices, STREAMS creates and initializes a `streamtab` structure for each end of the pipe. As with modules and drivers, the `streamtab` structure defines the pipe. The *st_rdinit*, however, points to the read `qinit` structure for the Stream head and not for a driver. Similarly, the *st_wdinit* points to the Stream head's write `qinit` structure and not to a driver. The *st_muxrinit* and *st_muxwinit* are initialized to null since a pipe cannot be a multiplexer driver.

When the `pipe` system call is executed, two Streams are created. Two Stream headers are created from `stdata` data structures and two Stream heads are created from two pairs of `queue` structures. The content of `stdata` and `queue` are initialized with the same values for all pipes.

Each Stream header represents one end of the pipe and points to the downstream half of each Stream head queue pair. Unlike STREAMS-based devices, however, the downstream portion of the Stream terminates at the upstream portion of the other Stream.

The *q_next* values are set so that the Stream head write `queue` points to the Stream head read `queue` on the other side. The *q_next* values for the Stream head's read `queue` are null since it terminates the Stream.

Adding and Removing Modules

As part of constructing a Stream, a module can be added (pushed) with an `ioctl I_PUSH` (see `streamio`(7)) system call. The push inserts a module beneath the Stream head. Because of the similarity of STREAMS components, the push operation is similar to the driver open. First, the address of the `qinit` structure for the module is obtained.

Next, STREAMS allocates a pair of `queue` structures and initializes their contents as in the driver open.

Then, *q_next* values are set and modified so that the module is put between the Stream head and its neighbor immediately downstream. Finally, the module open procedure (located via `qinit`) is called.

Each push of a module is independent, even in the same Stream. If the same module is pushed more than once on a Stream, there will be multiple occurrences of that module in the Stream. The total number of pushable modules that may be contained on any one Stream is limited by the kernel parameter nstrpush (see Appendix E, “Configuration").

An `ioctl I_POP` (see `streamio`(7)) system call removes (pops) the module immediately below the Stream head. The pop calls the module close procedure. On return from the module close, any messages left on the module’s message queues are freed (deallocated). Then, STREAMS connects the Stream head to the component previously below the popped module and releases the module’s `queue` pair. `I_PUSH` and `I_POP` enable a user process to dynamically alter the configuration of a Stream by pushing and popping modules as required. For example, a module may be removed and a new one inserted below the Stream head. Then the original module can be pushed back after the new module has been pushed.

Closing the Stream

The last `close` to a STREAMS file dismantles the Stream. Dismantling consists of popping any modules on the Stream and closing the driver. Before a module is popped, the `close` may delay to allow any messages on the write message queue of the module to be drained by module processing. Similarly, before the driver is closed, the `close` may delay to allow any messages on the write message queue of the driver to be drained by driver processing. If `O_NDELAY` (or `O_NONBLOCK`, see `open`(2)) is clear, `close` will wait up to 15 seconds for

each module to drain and up to 15 seconds for the driver to drain. The default close delay is 15 seconds, but this can be changed on a per-stream basis with the `I_SETCLTIME ioctl`.

Note – This delay is independent of any delay that the module or driver's close routine itself chooses to impose. If `O_NDELAY` (or `O_NONBLOCK`) is set, the pop is performed immediately and the driver is closed without delay. Messages can remain queued, for example, if flow control is inhibiting execution of the write queue `service` procedure. When all modules are popped and any wait for the driver to drain is completed, the driver close routine is called. On return from the driver close, any messages left on the driver's queues are freed, and the `queue` and `stdata` structures are released.

Note – STREAMS frees only the messages contained on a message queue. Any message or data structures used internally by the driver or module must be freed by the driver or module close procedure.

Stream Construction Example

The following example extends the previous communications device echoing example (see "Basic Streams Operations" on page 9) module in the Stream. The (hypothetical) module in this example can convert (change case, delete, duplicate) selected alphabetic characters.

Inserting Modules

An advantage of STREAMS over the traditional character I/O mechanism stems from the ability to insert various modules into a Stream to process and manipulate data that pass between a user process and the driver. In the example, the character conversion module is passed a command and a corresponding string of characters by the user. All data passing through the module are inspected for instances of characters in this string; the operation identified by the command is performed on all matching characters. The necessary declarations for this program are shown:

Code Example 3-1 Header Definition

```
#include <string.h>
#include <fcntl.h>
#include <stropts.h>
#define        BUFLEN         1024
/*
 * These definitions would typically be
 * found in a header file for the module
 */
#define XCASE               1 /* change alphabetic case of char */
#define    DELETE           2 /* delete char */
#define    DUPLICATE        3 /* duplicate char */
main()
{
    char buf[BUFLEN];
    int fd, count;
    struct strioctl strioctl;
```

The first step is to establish a Stream to the communications driver and insert the character conversion module. The following sequence of system calls accomplishes this (though not a runnable example):

Code Example 3-2 Pushing a Module

```
if ((fd = open("/dev/term/a", O_RDWR)) < 0) {
    perror("open failed");
    exit(1);
}
if (ioctl(fd, I_PUSH, "chconv") < 0) {
    perror("ioctl I_PUSH failed");
    exit(2);
}
```

The `I_PUSH ioctl` call directs the Stream head to insert the character conversion module between the driver and the Stream head, creating the Stream shown in Figure 3-3.

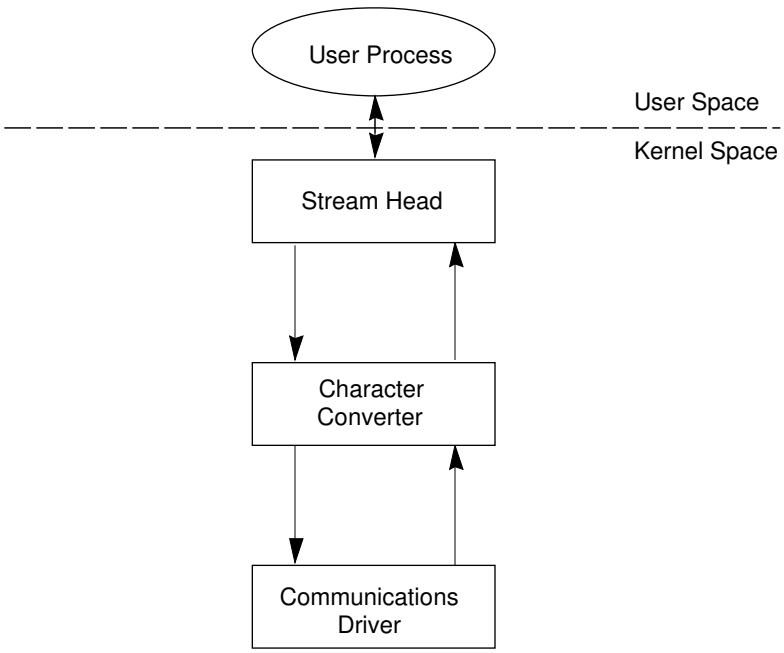

Figure 3-3 Case Converter Module

An important difference between STREAMS drivers and modules is illustrated here. Drivers are accessed through a node or nodes in the file system and may be opened just like any other device. Modules, on the other hand, do not occupy a file system node. Instead, they are identified through a separate naming convention, and are inserted into a Stream using `I_PUSH` or `autopush`. The name of a module is defined by the module developer.

Modules are pushed onto a Stream and removed from a Stream in Last-In-First-Out (LIFO) order. Therefore, if a second module was pushed onto this Stream, it would be inserted between the Stream head and the character conversion module.

Module and Driver Control

The next step in this example is to pass the commands and corresponding strings to the character conversion module. This can be accomplished by issuing `ioctl` calls to the character conversion module as shown in the next example.

The material from here to the end of the section is correct, but incomplete. The framework contains features that allow modules and drivers to process `ioctl`s without requiring user programs to first encapsulate them with `I_STR` (that is, the `ioctl`'s in the examples would look like `ioctl(fd,DELETE,"AEIOU");`).

This style of call works only for modules and drivers that have been converted to use the new facilities. Such modules and drivers will continue to accept the `I_STR` form, even after conversion.

Code Example 3-3 Processing Ioctl

```
/* change all uppercase vowels to lowercase */
strioctl.ic_cmd = XCASE;
strioctl.ic_timout = 0; /* default timeout (15 sec) */
strioctl.ic_dp = "AEIOU";
strioctl.ic_len = strlen(strioctl.ic_dp);
if (ioctl(fd, I_STR, &strioctl) < 0) {
        perror("ioctl I_STR failed");
        exit(3);
}
/* delete all instances of the chars 'x' and 'X' */
strioctl.ic_cmd = DELETE;
strioctl.ic_dp = "xX";
strioctl.ic_len = strlen(strioctl.ic_dp);
if (ioctl(fd, I_STR, &strioctl) < 0) {
        perror("ioctl I_STR failed");
        exit(4);
}
```

`ioctl` requests are issued to STREAMS drivers and modules indirectly, using the `I_STR` `ioctl` call (see `streamio`(7)). The argument to `I_STR` must be a pointer to a `strioctl` structure, which specifies the request to be made to a module or driver. This structure is defined in `streamio(7)` and has the following format:

```
struct strioctl {
    int         ic_cmd;             /* ioctl request */
    int         ic_timout;          /* ACK/NAK timeout */
    int         ic_len;             /* length of data argument */
    char        *ic_dp;             /* ptr to data argument */
};
```

where *ic_cmd* identifies the command intended for a module or driver, *ic_timout* specifies the number of seconds an `I_STR` request should wait for an acknowledgment before timing out, *ic_len* is the number of bytes of data to accompany the request, and *ic_dp* points to that data.

In the example, two separate commands are sent to the character-conversion module. The first sets *ic_cmd* to the command `XCASE` and sends as data the string "AEIOU"; it will convert all uppercase vowels in data passing through the module to lowercase. The second sets *ic_cmd* to the command `DELETE` and sends as data the string "xX"; it will delete all occurrences of the characters 'x' and 'X' from data passing through the module. For each command, the value of *ic_timout* is set to zero, which specifies the system default timeout value of 15 seconds. The *ic_dp* field points to the beginning of the data for each command; *ic_len* is set to the length of the data.

`I_STR` is intercepted by the Stream head, which packages it into a message, using information contained in the `strioctl` structure, and sends the message downstream. Any module that does not understand the command in *ic_cmd* should pass the message further downstream. The request will be processed by the module or driver closest to the Stream head that understands the command specified by *ic_cmd*. The `ioctl` call will block up to *ic_timout* seconds, waiting for the target module or driver to respond with either a positive or negative acknowledgment message. If an acknowledgment is not received in *ic_timout* seconds, the `ioctl` call will fail.

Note – Only one `ioctl` can be active on a Stream at one time, whether or not it is issued with `I_STR`. Further requests will block until the active `ioctl` is acknowledged and the system call concludes.

The `strioctl` structure is also used to retrieve the results, if any, of an `I_STR` request. If data is 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 remainder of this example is identical to the example in Chapter 2, “Overview of STREAMS".

Code Example 3-4 Process input

```
    while ((count = read(fd, buf, BUFLEN)) > 0) {
            if (write(fd, buf, count) != count) {
                perror("write failed");
                break;
            }
    }
    exit(0);
}
```

Notice that the character conversion processing was realized with *no* change to the communications driver.

The `exit` system call will dismantle the Stream before terminating the process. The character conversion module will be removed from the Stream automatically when it is closed. Alternatively, modules may be removed from a Stream using the `I_POP ioctl` call described in `streamio`(7). This call removes the topmost module on the Stream, and enables a user process to alter the configuration of a Stream dynamically, by popping modules as needed.

A few of the important `ioctl` requests supported by STREAMS have been discussed. Several other requests are available to support operations such as determining if a given module exists on the Stream, or flushing the data on a Stream. These requests are described fully in `streamio`(7).

3