STREAMS Programming Guide

ORACLE®

Contents

Using This Documentation

- **Overview** – The *STREAMS Programming Guide* describes how to use STREAMS in designing and implementing applications and STREAMS modules and drivers, for architectures that conform to the Oracle Solaris 7 DDI/DDK.
- **Audience** – This manual is a guide for application, driver, and module developers. The reader must know C programming in a UNIX® environment, and be familiar with the system interfaces. Driver and module developers should also be familiar with the book *Writing Device Drivers*.
- **Required knowledge** – Readers of this guide should be familiar with writing device drivers. Advanced knowledge of Oracle Solaris internals will be helpful.

Product Documentation Library

Late-breaking information and known issues for this product are included in the documentation library at http://www.oracle.com/pls/topic/lookup?ctx=E36784.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Feedback

Provide feedback about this documentation at http://www.oracle.com/goto/docfeedback.

PART   I

Application Programming Interface

Part I of this manual contains:

♦ ♦ ♦   **C H A P T E R   1**

1

Overview of STREAMS

This chapter provides a foundation for later chapters. Background and simple definitions are followed by an overview of the STREAMS mechanisms. Because the application developer is concerned with a different subset of STREAMS interfaces than the kernel-level developer, application and kernel levels are described separately.

- "STREAMS Definitions" on page 12
- "When to Use STREAMS" on page 15
- "How STREAMS Works—Application Interface" on page 15
- "How STREAMS Works at the Kernel Level" on page 17
- "Service Interfaces" on page 23

What Is STREAMS?

STREAMS is a general, flexible programming model for UNIX system communication services. STREAMS defines standard interfaces for character input/output (I/O) within the kernel, and between the kernel and the rest of the UNIX system. The mechanism consists of a set of system calls, kernel resources, and kernel routines.

STREAMS enables you to create modules to provide standard data communications services and then manipulate the modules on a stream. From the application level, modules can be dynamically selected and interconnected. No kernel programming, compiling, and link editing are required to create the interconnection.

STREAMS provides an effective environment for kernel services and drivers requiring modularity. STREAMS parallels the layering model found in networking protocols. For example, STREAMS is suitable for:

- Implementing network protocols
- Developing character device drivers
- Developing network controllers (for example, for an Ethernet card)
- I/O terminal services

The fundamental STREAMS unit is the *stream*. A stream is a full-duplex bidirectional data-transfer path between a process in user space and a STREAMS driver in kernel space. A stream has three parts: a stream head, zero or more modules, and a driver.

FIGURE   1-1   Simple Stream
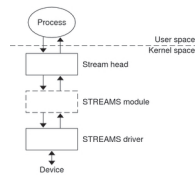
STREAMS Definitions

The capitalized word "STREAMS" refers to the STREAMS programming model and facilities. The word "stream" refers to an instance of a full-duplex path using the model and facilities between a user application and a driver.

Stream as a Data Path

A stream is a data path that passes data in both directions between a STREAMS driver in kernel space, and a process in user space. An application creates a stream by opening a STREAMS device (see Figure 1-1).

Stream Head

A stream head is the end of the stream nearest the user process. It is the interface between the stream and the user process. When a STREAMS device is first opened, the stream consists of only a stream head and a STREAMS driver.

STREAMS Module

A STREAMS module is a defined set of kernel-level routines and data structures. A module does "black-box" processing on data that passes through it. For example, a module converts lowercase characters to uppercase, or adds network routing information. A STREAMS module

is dynamically pushed on the stream from the user level by an application. Full details on modules and their operation are covered in Chapter 10, "STREAMS Modules".

STREAMS Device Driver

A STREAMS device driver is a character device driver that implements the STREAMS interface. A STREAMS device driver exists below the stream head and any modules. It can act on an external I/O device, or it can be an internal software driver, called a *pseudo-device driver*. The driver transfers data between the kernel and the device. The interfaces between the driver and kernel are known collectively as the Oracle Solaris operating environment Device Driver Interface/Driver Kernel Interface (Oracle Solaris operating environment DDI/DKI). The relationship between the driver and the rest of the UNIX kernel is explained in "Writing Device Drivers for Oracle Solaris 11.2 ". Details of device drivers are explained in Chapter 9, "STREAMS Drivers".

STREAMS Data

Data on a stream is passed in the form of messages. Messages are the means by which all I/O is done under STREAMS. Each stream head, STREAMS module, and driver has a *read side* and a *write side*. When messages go from one module's read side to the next module's read side, they are said to be traveling upstream. Messages passing from one module's write side to the next module's write side are said to be traveling downstream. Kernel-level operation of messages is discussed in "Message Components" on page 20 .

Message Queues

Each stream head, driver, and module has its own pair of queues, one queue for the read side and one queue for the write side. Messages are ordered into queues, generally on a first-in, first-out basis (FIFO), according to priorities associated with them. Kernel-level details of queues are covered in "Structure of a Message Queue" on page 21 .

FIGURE   1-2   Messages Passing Using Queues
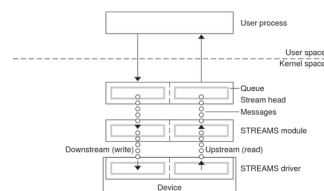
Communicating With a STREAMS Device

To communicate with a STREAMS device, an application's process uses read(2), write(2), getmsg(2), getpmsg(2), putmsg(2), putpmsg(2), and ioctl(2) to transmit or receive data on a stream.

From the command line, configure a stream with autopush(1M). From within an application, configure a stream with ioctl(2) as described in streamio(7I).

The ioctl(2) interface performs control operations on and through device drivers that cannot be done through the read(2) and write(2) interfaces. ioctl(2) operations include pushing and popping modules on and off the stream, flushing the stream, and manipulating signals and options. Certain ioctl(2) commands for STREAMS operate on the whole stream, not just the module or driver. The streamio(7I) manual page describes STREAMS ioctl(2) commands. Chapter 4, "Application Access to the STREAMS Driver and Module Interfaces" details interstream communications.

STREAMS Multiplexing

The modularity of STREAMS allows one or more upper streams to route data into one or more lower streams. This process is defined as multiplexing (mux). Example configurations of multiplexers are described in "Configuring Multiplexed Streams" on page 22 .

STREAMS Polling

Polling within STREAMS enables a user process to detect events occurring at the stream head, specifying the event to look for and the amount of time to wait for it to happen. An application might need to interact with multiple streams. The poll(2) system call enables applications to detect events that occur at the head of one or more streams. Chapter 3, "STREAMS Application-Level Mechanisms" describes polling.

Message Transfer Flow Control

Flow control regulates the rate of message transfer between the user process, stream head, modules, and driver. With flow control, a module that cannot process data at the rate being sent can queue the data to avoid flooding modules upstream. Flow control is local to each module or driver, and is voluntary. Chapter 8, "STREAMS Kernel-Level Mechanisms" describes flow control.

When to Use STREAMS

The STREAMS framework is most useful when modularity and configurability are issues. For instance, network drivers, terminal drivers, and graphics I/O device drivers benefit from using STREAMS. Modules can be pushed (added) and popped (removed) to create desired program behavior.

STREAMS is general enough to provide modularity between a range of protocols. It is a major component in networking support utilities for UNIX System V because it facilitates communication between network protocols.

How STREAMS Works—Application Interface

An application opens a STREAMS device, which creates the stream head to access the device driver. The stream head packages the data from the user process into STREAMS messages, and passes it downstream into kernel space. One or more cooperating modules can be pushed on a stream between the stream head and driver to customize the stream and perform any of a range of tasks on the data before passing it on. On the other hand, a stream might consist solely of the stream head and driver, with no module at all.

Opening a Stream

To a user application, a STREAMS device resembles an ordinary character I/O device, as it has one or more nodes associated with it in the file system, and is opened by calling open(2).

The file system represents each device as a special file. There is an entry in the file for the major device number, identifying the actual device driver that will activate the device. There are corresponding separate minor device numbers for each instance of a particular device, for example, for a particular port on a serial card, or a specific pseudo-terminal such as those used by a windowing application.

Different minor devices of a driver cause a separate stream to be connected between a user process and the driver. The first open call creates the stream; subsequent open calls respond with a file descriptor referencing that stream. If the same minor device is opened more than once, only one stream is created.

However, drivers can support a user process getting a dedicated stream without the application distinguishing which minor device is used. In this case, the driver selects any unused minor device to be used by the application. This special use of a minor device is called *cloning*. Chapter 9, "STREAMS Drivers" describes properties and behavior of clone devices.

Once a device is opened, a user process can send data to the device by calling write(2), and receive data from the device by calling read(2). Access to STREAMS drivers using read

and write is compatible with the traditional character I/O mechanism. STREAMS-specific applications also can call getmsg(2), getpmsg(2), putmsg(2), and putpmsg(2) to pass data to and from the stream.

Closing a Stream

The close(2) interface closes a device and dismantles the associated stream when the last open reference to the stream is closed. The exit(2) interface terminates the user process and closes all open files.

Controlling Data Flow

If the stream exerts flow control, the write(2) call blocks until flow control has been relieved, unless the file has been specifically advised not to. open(2) or fcntl(2) can be used to control this nonblocking behavior.

Simple Stream Example

Example 1-1 shows how an application might use a simple stream. Here, the user program interacts with a communications device that provides point-to-point data transfer between two computers. Data written to the device is transmitted over the communications line, and data arriving on the line is retrieved by reading from the device.

EXAMPLE 1-1   Simple Stream

```
#include <sys/fcntl.h>
#include <stdio.h>

main()
{
  char buf[1024];
  int fd, count;

  if ((fd = open("/dev/ttya", O_RDWR)) < 0) {
   perror("open failed");
   exit(1);
  }
  while ((count = read(fd, buf, sizeof(buf))) > 0) {
   if (write(fd, buf, count) != count) {
    perror("write failed");
    break;
```

```
        }
      }
     exit(0);
}
```

In this example, `/dev/ttya` identifies an instance of a serial communications device driver. When this file is opened, the system recognizes the device as a STREAMS device and connects a stream to the driver. Figure 1-3 shows the state of the stream following the call to open(2).

FIGURE 1-3   Stream to Communications Driver
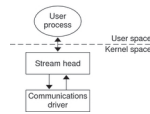
This example illustrates a simple loop, with the application reading data from the communications device, then writing the input back to the same device, echoing all input back over the communications line. The program reads up to 1024 bytes at a time, and then writes the number of bytes just read.

read(2) returns the available data, which can contain fewer than 1024 bytes. If no data is currently available at the stream head, read(2) blocks until data arrives.

Note - The application program must loop on read(2) until the desired number of bytes are read. The responsibility for the application getting all the bytes it needs is that of the application developer, not the STREAMS facilities.

Similarly, the write(2) call attempts to send the specified number of bytes to `/dev/ttya`. The driver can implement a flow-control mechanism that prevents a user from exhausting system resources by flooding a device driver with data.

How STREAMS Works at the Kernel Level

Developers implementing STREAMS device drivers and STREAMS modules use a set of STREAMS-specific functions and data structures. This section describes some basic kernel-level STREAMS concepts.

Creating the Stream Head

The stream head is created when a user process opens a STREAMS device. It translates the interface calls of the user process into STREAMS messages, which it sends to the stream. The stream head also translates messages originating from the stream into a form that the application can process. The stream head contains a pair of queues; one queue passes messages upstream from the driver, and the other passes messages to the driver. The queues are the pipelines of the stream, passing data between the stream head, modules, and driver.

Message Processing

A STREAMS module does processing operations on messages passing from a stream head to a driver or from a driver to a stream head. For example, a TCP module might add header information to the front of data passing downstream through it. Not every stream requires a module. There can be zero or more modules in a stream.

Modules are stacked (pushed) onto and unstacked (popped) from a stream. Each module must provide `open()`, `close()`, and `put()` entries and provides a `service()` entry if the module supports flow control.

Like the stream head, each module contains a pair of queue structures, although a module only queues data if it is implementing flow control. Figure 1-4 shows the queue structures Au/Ad associated with Module A ("u" for upstream "d" for downstream) and Bu/Bd associated with Module B.

The two queues operate completely independently. Messages and data can be shared between upstream and downstream queues only if the module functions are specifically programmed to share data.

Within a module, one queue can refer to the messages and data of the opposing queue. A queue can directly refer to the queue of the successor module (adjacent in the direction of message flow). For example, in Figure 1-4 , Au (the upstream queue from Module A) can reference Bu (the upstream queue from Module B). Similarly Queue Bd can reference Queue Ad.

FIGURE   1-4   Stream in More Detail
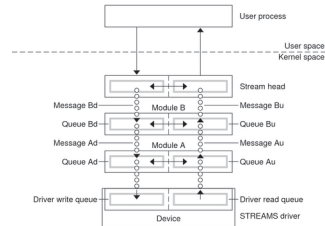
Both queues in a module contain messages, processing procedures, and private data.

Messages   Blocks of data that pass through, and can be operated on by, a module.

Processing   Individual `put` and `service` routines on the read and write queues
procedures   process messages. The `put` procedure passes messages from one queue to
                        the next in a stream and is required for each queue. It can do additional
                        message processing. The `service` procedure is optional and does
                        deferred processing of messages. These procedures can send messages
                        either upstream or downstream. Both procedures can also modify the
                        private data in their module.

Private data   Data private to the module (for example, state information and translation
                        tables).

Open and close   Entry points must be provided. The `open` routine is invoked when the
                        module is pushed onto the stream or the stream is reopened. The `close` is
                        invoked when the module is popped or the stream is closed.

A module is initialized by either an `I_PUSH` ioctl(2), or pushed automatically during an

open if a stream has been configured by the autopush(1M) mechanism, or if that stream is
reopened.

A module is disengaged by `close` or the `I_POP` ioctl(2).

Structure of a STREAMS Device Driver

STREAMS device drivers are structurally similar to STREAMS modules and character device
drivers. The STREAMS interfaces to driver routines are identical to the interfaces used for
modules. For instance they must both declare `open`, `close`, `put`, and `service` entry points.

There are some significant differences between modules and drivers.

A driver:

- Must be able to handle interrupts from the device.
- Is represented in file system by a character-special file.
- Is initialized and disengaged using `open(2)` and `close(2)`. `open(2)` is called when the device is first opened and for each reopen of the device. `close(2)` is only called when the last reference to the stream is closed.

Both drivers and modules can pass signals, error codes, and return values to processes using message types provided for that purpose.

Message Components

All kernel-level input and output under STREAMS is based on messages. STREAMS messages are built in sets of three:

- a message header structure (`msgb(9S)`) that identifies the message instance.
- a data block structure (`datab(9S)`) points to the data of the message.
- the data itself

Each data block and data pair can be referenced by one or more message headers. The objects passed between STREAMS modules are pointers to messages. Messages are sent through a stream by successive calls to the `put` procedure of each module or driver in the stream. Messages can exist as independent units, or on a linked list of messages called a *message queue*. STREAMS utility routines enable developers to manipulate messages and message queues.

All STREAMS messages are assigned message types to indicate how they will be used by modules and drivers and how they will be handled by the stream head. Message types are assigned by the stream head, driver, or module when the message is created. The stream head converts the system calls `read`, `write`, `putmsg`, and `putpmsg` into specified message types, and sends them downstream. It responds to other calls by copying the contents of certain message types that were sent upstream.

Message Queueing Priority

Sometimes messages with urgent information, such as a break or alarm conditions, must pass through the stream quickly. To accommodate them, STREAMS uses message queuing priority, and high-priority message types. All messages have an associated priority field. Normal (ordinary) messages have a priority of zero, while priority messages have a priority band greater than zero. High-priority messages have a high priority by virtue of their message type, are not

blocked by STREAMS flow control, and are processed ahead of all ordinary messages on the queue.

Nonpriority, ordinary messages are placed at the end of the queue following all other messages that can be waiting. Priority messages can be either high priority or priority band messages. High-priority messages are placed at the head of the queue but after any other high-priority messages already in the queue. Priority band messages enable support of urgent, expedited data. Priority band messages are placed in the queue in the following order:

- after high-priority messages but before ordinary messages.
- below all messages that have a priority greater than or equal to their own.
- above any messages with a lesser priority.

Figure 1-5 shows the message queueing priorities.

FIGURE   1-5   Message Priorities
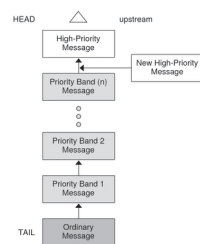
High-priority message types cannot be changed into normal or priority band message types. Certain message types come in equivalent high-priority or ordinary pairs (for example, M_PCPROTO and M_PROTO), so that a module or device driver can choose between the two priorities when sending information.

Structure of a Message Queue

A *queue* is an interface between a STREAMS driver or module and the rest of the stream (see queue(9S)). The queue structure holds the messages, and points to the STREAMS processing routines that should be applied to a message as it travels through a module. STREAMS modules and drivers must explicitly place messages on a queue, for example, when flow control is used.

Each open driver or pushed module has a pair of queues allocated, one for the read side and one for the write side. Queues are always allocated in pairs. Kernel routines are available to access each queue's mate. The queue's put or service procedure can add a message to the current

queue. If a module does not need to queue messages, its `put` procedure can call the neighboring queue's `put` procedure.

The queue's `service` procedure deals with messages on the queue, usually by removing successive messages from the queue, processing them, and calling the `put` procedure of the next module in the stream to pass the message to the next queue. Chapter 7, "STREAMS Framework – Kernel Level" discusses the `service` and `put` procedures in more detail.

Each queue also has a pointer to an `open` and `close` routine. The `open` routine of a driver is called when the driver is first opened and on every successive open of the stream. The `open` routine of a module is called when the module is first pushed on the stream and on every successive open of the stream. The `close` routine of the module is called when the module is popped (removed) off the stream, or at the time of the final close. The `close` routine of the driver is called when the last reference to the stream is closed and the stream is dismantled.

Configuring Multiplexed Streams

Previously, streams were described as stacks of modules, with each module (except the head) connected to one upstream module and one downstream module. While this can be suitable for many applications, others need the ability to multiplex streams in a variety of configurations. Typical examples are terminal window facilities, and internetworking protocols (that might route data over several subnetworks).

An example of a multiplexer is a module that multiplexes data from several upper streams to a single lower stream. An upper stream is one that is upstream from the multiplexer, and a lower stream is one that is downstream from the multiplexer. A terminal windowing facility might be implemented in this fashion, where each upper stream is associated with a separate window.

A second type of multiplexer might route data from a single upper stream to one of several lower streams. An internetworking protocol could take this form, where each lower stream links the protocol to a different physical network.

A third type of multiplexer might route data from one of many upper streams to one of many lower streams.

The STREAMS mechanism supports the multiplexing of streams through special pseudo-device drivers. A user can activate a linking facility mechanism within the STREAMS framework to dynamically build, maintain, and dismantle multiplexed stream configurations. Simple configurations like those shown previously can be combined to form complex, multilevel multiplexed stream configurations.

STREAMS multiplexing configurations are created in the kernel by interconnecting multiple streams. Conceptually, a multiplexer can be divided into two components—the upper multiplexer and the lower multiplexer. The lower multiplexer acts as a stream head for one or more lower streams. The upper multiplexer acts as a device for one or more upper streams. How

data is passed between the upper and lower multiplexer is up to the implementation. Chapter 13, "STREAMS Multiplex Drivers" covers implementing multiplexers.

Multithreading the Kernel

The Oracle Solaris operating environment kernel is multithreaded to make effective use of symmetric shared-memory multiprocessor computers. All parts of the kernel, including STREAMS modules and drivers, must ensure data integrity in a multiprocessing environment. For the most part, developers must ensure that concurrently running kernel threads do not attempt to manipulate the same data at the same time. The STREAMS framework provides multithreaded (MT) STREAMS perimeters, which provides the developer with control over the level of concurrency allowed in a module. The DDI/DKI provides several advisory locks for protecting data. See Chapter 12, "Multithreaded STREAMS" for more information.

Service Interfaces

Using STREAMS, you can create modules that present a service interface to any neighboring module or device driver, or between the top module and a user application. A service interface is defined in the boundary between two neighbors.

In STREAMS, a service interface is a set of messages and the rules that allow these messages to pass across the boundary. A module using a service interface, for example, receives a message from a neighbor and responds with an appropriate action (perhaps sending back a request to retransmit) depending on the circumstances.

You can stack a module anywhere in a stream, but connecting sequences of modules with compatible protocol service interfaces is better. For example, a module that implements an X.25 protocol layer, as shown in Figure 1-6 , presents a protocol service interface at its input and output sides. In this case, other modules should be connected to the input and output side if they have the compatible X.25 service interface only.

Manipulating Modules

With STREAMS, you can manipulate modules from the user application level, interchange modules with common service interfaces, and change the service interface to a STREAMS user process. These capabilities yield further benefits when working with networking services and protocols:

- User-level programs can be independent of underlying protocols and physical communication media.

■ Network architectures and higher-level protocols can be independent of underlying protocols, drivers, and physical communication media.

■ Higher-level services can be created by selecting and connecting lower-level services and protocols.

The following examples show the benefits of STREAMS capabilities for creating service interfaces and manipulating modules. These examples are only illustrations and do not necessarily reflect real situations.

Protocol Portability

Figure 1-6 shows how an X.25 protocol module can work with different drivers on different machines by using compatible service interfaces. The X.25 protocol module interfaces are Connection Oriented Network Service (CONS) and Link Access Protocol – Balanced (LAPB).

FIGURE   1-6   Protocol Module Portability
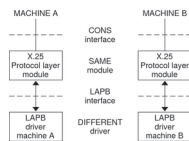
Protocol Substitution

You can alternate protocol modules and device drivers on a system if the alternates are implemented to an equivalent service interface.

Protocol Migration

Figure 1-7 shows how STREAMS can move functions between kernel software and front-end firmware. A common downstream service interface lets the transport protocol module be independent of the number or type of modules below it. The same transport module will connect without modification to either an X.25 module or X.25 driver that has the same service interface.

By shifting functions between software and firmware, you can produce cost-effective, functionally equivalent systems over a wide range of configurations. This means you can swiftly incorporate technological advances. The same transport protocol module can be used on

a lower-capacity machine, where economics preclude the use of front-end hardware, and also on a larger scale system where a front-end is economically justified.

FIGURE 1-7 Protocol Migration
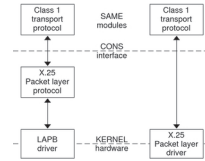
Module Reusability

Figure 1-8 shows the same canonical module (for example, one that provides delete and kill processing on character strings) reused in two different streams. This module would typically be implemented as a filter, with no service interface. In both cases, a TTY interface is presented to the stream's user process because the module is nearest the stream head.

FIGURE 1-8 Module Reusability

♦♦♦ **C H A P T E R 2**

2
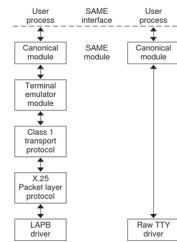
STREAMS Application-Level Components

This chapter shows how to construct, use, and dismantle a stream using STREAMS-related system calls. It provides a general discussion of the relationship between STREAMS components in a simple streams example.

- "STREAMS Interfaces" on page 27
- "Opening a STREAMS Device File" on page 28
- "Queue Allocation" on page 30
- "Adding and Removing Modules" on page 30
- "Closing the Stream" on page 30
- "Stream Construction Example" on page 31

STREAMS Interfaces

The stream head provides the interface between the stream and an application program. After a stream has been opened, STREAMS-related system calls enable a user process to insert and delete (push and pop) modules. That process can then communicate with and control the operation of the stream head, 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.

STREAMS System Calls

Table 2-1 offers an overview of some basic STREAMS-related system calls.

TABLE 2-1   Summary of Basic STREAMS-related System Calls

Function	Description
open(2)	Opens a stream
close(2)	Closes a stream
read(2)	Reads data from a stream

Function	Description
write(2)	Writes data to a stream
ioctl(2)	Controls a stream
getmsg(2)	Receives a message at the stream head
getpmsg(2)	Receives a priority message at the stream head
putmsg(2)	Sends a message downstream
putpmsg(2)	Sends a priority message downstream
poll(2)	Identifies files on which a user can send or receive messages, or on which certain events have occurred (not restricted to STREAMS, although historically it was)
pipe(2)	Creates a bidirectional channel that provides a communication path between multiple processes

Note - Sections 1, 2, 3, 7, and 9 of the online manual pages (man pages) contain all the STREAMS information.

Action Summary

The open(2) system call recognizes a STREAMS special 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 way as with traditional character files. ioctl(2) lets users perform functions specific to a particular device. STREAMS ioctl(2) commands (see the streamio(7I) man page) support a variety of functions to access and control streams. The final close(2) on a stream dismantles it.

The poll(2) system call provides a mechanism for multiplexing input/output over a set of file descriptors that reference open files. putmsg(2) and getmsg(2) and the putpmsg(2) and getpmsg(2) send and receive STREAMS messages, and can act on STREAMS modules and drivers through a service interface.

Opening a STREAMS Device File

One way to construct a stream is to call open(2) to open a STREAMS special file. If the open call is the initial file open, a stream is created. (There is one stream for each major or minor

device pair.) If this open is not the initial open of this stream, the open procedures of the driver and all pushable modules on the stream are called.

Sometimes a user process needs to connect a new stream to a driver regardless of which minor device is used to access the driver. Instead of the user process polling for an available minor device node, STREAMS provides a facility called *clone open.* If a STREAMS driver is implemented as a clone device, a single node in the file system may be opened to access any unused device that the driver controls. This special node guarantees that the user is allocated a separate stream to the driver for every open call. Each stream is associated with an unused major or minor device, so the total number of streams that can connect to a particular clone driver is limited to the number of minor devices configured for the driver.

Clone devices are used, for example, in a networking environment where a protocol pseudo-device driver requires each user to open a separate stream to establish communication.

You can open a clone device in two ways. The first is to create a node with the major number of the clone device (--) and a minor number corresponding to the major number of the device to be cloned. For example /dev/ps0 might have a major number of 50 and a minor number of 0 for normal opens. The clone device may have a major number of 40. By creating a node /dev/ps with a major number of 40 and a minor number of 50, a clonable device is created. In this case, the driver is passed a special flag (CLONEOPEN) that tells it to return a unique minor device number.

The second way is to have the driver open itself as a clone device, that is, the driver returns a unique minor number.

When a stream is already open, further opens of the same device call the open routines of all modules and the driver on that stream. In this case, 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 is called, followed by the open routine of the driver. This is opposite to the initial order of opens when the stream is created.

STREAMS also offers autopush. On 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(2) returns. For more information see the autopush(1M) and sad(7D) man pages.

Initializing Details

There is one stream head per stream. The stream head, which is initiated by the first open call, is created from a data structure and a pair of queue structures. The content of the stream head and queues is initialized with predetermined values, including the stream head processing procedures.

Queue Allocation

STREAMS queues are allocated in pairs. One queue is always the upstream or read-side; the other is the downstream or write-side. These queues hold the messages, and tell the kernel which processing routines apply to each message passing through a module. The queue structure type is queue_t. Fields in the queue data structure are detailed in queue(9S).

Adding and Removing Modules

As part of constructing a stream, a module can be added (pushed) with an I_PUSH ioctl(2) (see streamio(7I)) 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.

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 are multiple occurrences of the 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.

An I_POP ioctl(2) (see streamio(7I)) 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). The stream head then connects 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.

You can also restrict which modules can be popped with I_POP calls by placing an anchor in the stream at any module you want to "lock down." The I_ANCHOR ioctl prevents a module from being popped except by a privileged process. See "STREAMS Anchors" on page 222 for more information about working with anchors.

Closing the Stream

The last close to a STREAMS device dismantles the stream. Dismantling consists of popping any modules on the stream and closing the driver. Before a module is popped, the close(2) 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(2) 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) is clear, close(2) waits up to 15 seconds for each module to drain and up to 15 seconds for the driver to drain, see open(2) and fcntl(2). The default close delay is 15 seconds, but this can be changed on a per-stream basis with the I_SETCLTIME ioctl(2).

The close 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 stream head structures are released.

Stream Construction Example

This example extends the communications device-echoing example shown in "Simple Stream Example" on page 16. The module in this example converts (change case, delete, duplicate) selected alphabetic characters.

Note - The complete listing of the module is on the CD.

Inserting Modules

An application can 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 receives a command and a corresponding string of characters from the user. All data passing through the module is inspected for instances of characters in this string. Whatever operation the command requires is performed on all characters that match the string.

EXAMPLE 2-1   Module Header File 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. This is accomplished by first opening (fd = open) then calling ioctl(2) to push the chconv module, as shown in the sequence of system calls in Example 2-2 .

EXAMPLE 2-2   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(2) call directs the stream head to insert the character conversion module between the driver and the stream head. The example illustrates an important difference between STREAMS drivers and modules. Drivers are accessed through a node or nodes in the file system (in this case /dev/term/a) and are opened just like other devices. Modules, on the other hand, are not devices. Identify modules through a separate naming convention, and insert them into a stream using I_PUSH or autopush. Figure 2-1 shows creation of the stream.

FIGURE 2-1   Pushing the Character Conversion Module
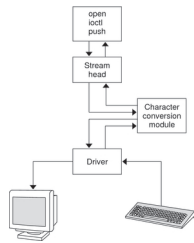
Modules are stacked onto a stream and removed from a stream in last-in, first-out (LIFO) order. Therefore, if a second module is pushed onto this stream, it is inserted between the stream head and the character conversion module.

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 calling ioctl(2) to invoke the character conversion module.

Example 2-3 uses the conventional I_STR ioctl(2), an indirect way of passing commands and data pointers. Example 2-4 shows the data structure for I_STR.

Instead of I_STR, some systems support transparent ioctls in which calls can be made directly. For example, a module calls I_PUSH. Both modules and drivers can process ioctls without requiring user programs to first encapsulate them with I_STR (that is, the ioctls 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 that also accept the I_STR form.

EXAMPLE 2-3   Processing ioctl(2)

```
/* 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);
}
```

In Example 2-3 , the module changes all uppercase vowels to lowercase, and deletes all instances of either uppercase or lowercase "x". ioctl(2) requests are issued indirectly, using I_STR ioctl(2) (see streamio(7I)). 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 described in streamio(7I) and has the format shown in the following example.

EXAMPLE 2-4   strioctl Structure

```
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 acknowledgement before timing out.
`ic_len`	The number of bytes of data to accompany the request.
`ic_dp`	Points to the data. In the example, two separate commands are sent to the character-conversion module:

- The first command sets `ic_cmd` to the command `XCASE` and sends as data the string "AEIOU." It converts all uppercase vowels in data passing through the module to lowercase.

- The second command sets `ic_cmd` to the command `DELETE` and sends as data the string "xX." It deletes 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. `ic_dp` 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, then sends the message downstream. Any module that cannot process the command in `ic_cmd` passes the message further downstream. The request is processed by the module or driver closest to the stream head that understands the command specified by `ic_cmd`. ioctl(2) blocks up to `ic_timout` seconds, waiting for the target module or driver to respond with either a positive or negative acknowledgement message. If an acknowledgement is not received in `ic_timout` seconds, ioctl(2) fails.

Note - Only one ioctl(2) can be active on a stream at one time, regardless of whether it is issued with `I_STR`. Further requests will block until the active ioctl(2) 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` is set on return to indicate the amount of data returned. The remainder of this example is identical to Example 1-1 in Chapter 1, "Overview of STREAMS".

EXAMPLE 2-5   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.

exit(2) dismantles the stream before terminating the process. The character conversion module is removed from the stream automatically when it is closed. Alternatively, remove modules from a stream using I_POP ioctl(2) which is described in streamio(7I). 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.

Several other ioctl(2) requests support STREAMS operations, such as determining if a given module is on a stream, or flushing the data on a stream. streamio(7I) describes these requests.

3

STREAMS Application-Level Mechanisms

The previous chapters described the components of a stream from an application level. This chapter explains how those components work together. It shows how the kernel interprets system calls being passed from an application, so that driver and module developers can know what structures are being passed.

Message Handling

Messages are the communication medium between the user application process and the various components of the stream. This chapter describes the path they travel and the changes that occur to them. Chapter 8, "STREAMS Kernel-Level Mechanisms" covers the underlying mechanics of the kernel.

Modifying Messages

The `put`(9E) and `srv`(9E) interfaces process messages as they pass through the queue. Messages are generally processed by type, resulting in a modified message, one or more new messages, or no message at all. The message usually continues in the same direction it was passing through the queue, but can be sent in either direction. A `put`(9E) procedure can place messages on its queue as they arrive, for later processing by the `srv`(9E) procedure. For a more detailed explanation of `put`(9E) and `srv`(9E), see Chapter 8, "STREAMS Kernel-Level Mechanisms" .

Some kernel operations are explained here to show you how to manipulate the driver or module appropriately.

Message Types

STREAMS messages differ according to their intended purpose and their queueing priority. The contents of certain message types can be transferred between a process and a stream using system calls. Appendix A, "Message Types" describes message types in detail.

Control of Stream Head Processing

The stream head responds to a message by altering the processing associated with certain system calls. Six stream head characteristics can be modified. Four characteristics correspond to fields contained in queue (packet sizes — minimum and maximum, and watermarks — high and low). Packet sizes are discussed in this chapter. Watermarks are discussed in "Flush Handling" on page 65 in Chapter 4, "Application Access to the STREAMS Driver and Module Interfaces".

Read Options

The read options (so_readopt) specify two sets of three modes that can be set by the I_SRDOPT ioctl(2) (see streamio(7I)). Byte-stream mode approximately models pipe data transfer. Message nondiscard mode is similar to a TTY in canonical mode.

The first set of bits, RMODEMASK, deals with data and message boundaries, as shown in Table 3-1.

TABLE 3-1   Data and Message Boundaries

RMODEMASK		
RNORM	RMSGN	RMSGD

Byte-stream (RNORM)	The read(2) call finishes when the byte count is satisfied, the stream head read queue becomes empty, or a zero length message is encountered. A zero length message is put back in the queue. A subsequent read returns 0 bytes.
Message non-discard (RMSGN)	The read(2) call finishes when the byte count is satisfied or a message boundary is found, whichever comes first. Any data remaining in the message is put back on the stream head read queue.
Message discard (RMSGD)	The read(2) call finishes when the byte count is satisfied or a message boundary is found. Any data remaining in the message is discarded up to the message boundary.

The second set of bits, RPROTMASK, specifies the treatment of protocol messages by the read(2) system call as shown in Table 3-2.

TABLE 3-2   How read(2) Treats Protocol Messages

RPROTMASK		
RPROTNORM	RPROTDIS	RPROTDATA

Normal protocol (RPROTNORM)	The read(2) call fails with EBADMSG if an M_PROTO or M_PCPROTO message is at the front of the stream head read queue. This is the default operation protocol.
Protocol discard (RPROTDIS)	The read(2) call discards any M_PROTO or M_PCPROTO blocks in a message, delivering the M_DATA blocks to the user.
Protocol data (RPROTDAT)	The read(2) call converts the M_PROTO and M_PCPROTO message blocks to M_DATA blocks, treating the entire message as data.

Write Options

Send zero (I_SWROPT)	The write(2) mode is set using the value of the argument *arg*. Legal bit settings for *arg* are: SNDZERO—Send a zero-length message downstream when the write of 0 bytes occurs. To avoid sending a zero-length message when a write of 0 bytes occurs, this bit must not be set in *arg*. On failure, errno can be set to EINVAL—*arg* is above the legal value.

Message Queueing and Priorities

Any delay in processing messages causes message queues to grow. Normally, queued messages are handled in a first-in, first-out (FIFO) manner. However, certain conditions can require that associated messages (for instance, an error message) reach their stream destination as rapidly as possible. For this reason messages are assigned priorities using a priority band associated with each message. Ordinary messages have a priority of zero. High-priority messages are high priority by nature of their message type. Their priority band is ignored. By convention, they are not affected by flow control. Figure 3-1 shows how messages are ordered in a queue according to priority.

FIGURE   3-1   Message Ordering in a Queue
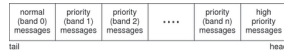
When a message is queued, it is placed after the messages of the same priority already in the queue (in other words, FIFO within their order of queueing). This affects the flow-control parameters associated with the band of the same priority. Message priorities range from 0 (normal) to 255 (highest). This provides up to 256 bands of message flow within a stream. Expedited data can be implemented with one extra band of flow (priority band 1) of data. This is shown in Figure 3-2 .

FIGURE   3-2   Message Ordering With One Priority Band
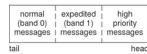
Controlling Data Flow and Priorities

The `I_FLUSHBAND`, `I_CKBAND`, `I_GETBAND`, `I_CANPUT`, and `I_ATMARK` `ioctl(2)`s support multiple bands of data flow. The `I_FLUSHBAND` `ioctl(2)` allows a user to flush a particular band of messages. "Flush Handling" on page 65 discusses it in more detail.

The `I_CKBAND` `ioctl(2)` checks if a message of a given priority exists on the stream head read queue. Its interface is:

```
ioctl (fd, I_CKBAND, pri);
```

The call returns 1 if a message of priority `pri` exists on the stream head read queue and 0 if no message of priority `pri` exists. If an error occurs, -1 is returned. Note that `pri` should be of type `int`.

The `I_GETBAND` `ioctl(2)` checks the priority of the first message on the stream head read queue. The interface is:

```
ioctl (fd, I_GETBAND, prip);
```

The call results in the integer referenced by `prip` being set to the priority band of the message on the front of the stream head read queue.

The `I_CANPUT` `ioctl(2)` checks if a certain band is writable. Its interface is:

```
ioctl (fd, I_CANPUT, pri);
```

The return value is 0 if the priority band `pri` is flow controlled, 1 if the band is writable, and -1 on error.

A module or driver can mark a message. This supports the ability of the Transmission Control Protocol (TCP) to indicate to the user the last byte of out-of-band data. Once marked, a message sent to the stream head causes the stream head to remember the message. A user can check whether the message on the front of its stream head read queue is marked with the `I_ATMARK` `ioctl`(2). If a user is reading data from the stream head, there are multiple messages on the read queue, and one of those messages is marked, the `read`(2) terminates when it reaches the marked message and returns the data only up to the marked message. Successive reads can return the rest of the data. Chapter 4, "Application Access to the STREAMS Driver and Module Interfaces" discusses this in more detail.

The `I_ATMARK` `ioctl`(2) has the format:

```
ioctl (fd, I_ATMARK, flag);
```

where `flag` can be either `ANYMARK` or `LASTMARK`. `ANYMARK` indicates that the user wants to check whether any message is marked. `LASTMARK` indicates that the user wants to see whether the message is the one and only one marked in the queue. If the test succeeds, 1 is returned. On failure, 0 is returned. If an error occurs, -1 is returned.

Accessing the Service Provider

The first routine presented, `inter_open`, opens the protocol driver device file specified by `path` and binds the protocol address contained in `addr` so that it can receive data. On success, the routine returns the file descriptor associated with the open stream; on failure, it returns -1 and sets `errno` to indicate the appropriate UNIX system error value. Example 3-1 shows the `inter_open` routine.

EXAMPLE 3-1   `inter_open` Routine

```
inter_open (char *path, oflags, addr)
{
  int fd;
  struct bind_req bind_req;
  struct strbuf ctlbuf;
  union  primitives rcvbuf;
  struct error_ack *error_ack;
  int flags;

  if ((fd = open(path, oflags)) < 0)
   return(-1);

 /* send bind request msg down stream */
```

```
    bind_req.PRIM_type = BIND_REQ;
    bind_req.BIND_addr = addr;
    ctlbuf.len = sizeof(struct bind_req);
    ctlbuf.buf = (char *)&bind_req;

    if (putmsg(fd, &ctlbuf, NULL, 0) < 0) {
     close(fd);
     return(-1);
    }
}
```

After opening the protocol driver, `inter_open` packages a bind request message to send downstream. `putmsg` is called to send the request to the service provider. The bind request message contains a control part that holds a `bind_req` structure, but it has no data part. `ctlbuf` is a structure of type `strbuf`, and it is initialized with the primitive type and address. Notice that the `maxlen` field of `ctlbuf` is not set before calling `putmsg`. That is because `putmsg` ignores this field. The `dataptr` argument to `putmsg` is set to `NULL` to indicate that the message contains no data part. The `flags` argument is 0, which specifies that the message is not a high-priority message.

After `inter_open` sends the bind request, it must wait for an acknowledgement from the service provider, as Example 3-2 shows.

EXAMPLE 3-2 Service Provider

```
/* wait for ack of request */

 ctlbuf.maxlen = sizeof(union primitives);
 ctlbuf.len = 0;
 ctlbuf.buf = (char *)&rcvbuf;
 flags = RS_HIPRI;

 if (getmsg(fd, &ctlbuf, NULL, &flags) < 0) {
  close(fd);
  return(-1);
}

 /* did we get enough to determine type? */
 if (ctlbuf.len < sizeof(long)) {
  close(fd);
  errno = EPROTO;
  return(-1);
}

 /* switch on type (first long in rcvbuf) */
  switch(rcvbuf.type) {
  default:
    close(fd);
    errno = EPROTO;
    return(-1);

  case OK_ACK:
```

```
    return(fd);

 case ERROR_ACK:
    if (ctlbuf.len < sizeof(struct error_ack)) {
     close(fd);
     errno = EPROTO;
     return(-1);
    }
    error_ack = (struct error_ack *)&rcvbuf;
    close(fd);
    errno = error_ack->UNIX_error;
    return(-1);
 }
}
```

getmsg is called to retrieve the acknowledgement of the bind request. The acknowledgement message consists of a control part that contains either an OK_ACK or an error_ack structure, and no data part.

The acknowledgement primitives are defined as high-priority messages. Messages are queued in a first-in, first-out (FIFO) manner within their priority at the stream head. The STREAMS mechanism allows only one high-priority message per stream at the stream head at one time. Any additional high-priority messages are discarded on reaching the stream head. (There can be only one high-priority message present on the stream head read queue at any time.) High-priority messages are particularly suitable for acknowledging service requests when the acknowledgement should be placed ahead of any other messages at the stream head.

Before calling getmsg, this routine must initialize the strbuf structure for the control part. buf should point to a buffer large enough to hold the expected control part, and maxlen must be set to indicate the maximum number of bytes this buffer can hold.

Because neither acknowledgement primitive contains a data part, the dataptr argument to getmsg is set to NULL. The flagsp argument points to an integer containing the value RS_HIPRI. This flag indicates that getmsg should wait for a STREAMS high-priority message before returning. This catches the acknowledgement primitives that are priority messages. Otherwise, if the flag is zero, the first message is taken. With RS_HIPRI set, even if a normal message is available, getmsg blocks until a high-priority message arrives.

On return from getmsg, check the len field to ensure that the control part of the retrieved message is an appropriate size. The example then checks the primitive type and takes appropriate actions. An OK_ACK indicates a successful bind operation, and inter_open returns the file descriptor of the open stream. An error_ack indicates a bind failure, and errno is set to identify the problem with the request.

Closing the Service Provider

The next routine in the service interface library example is inter_close, which closes the stream to the service provider.

```
inter_close(fd)
{
  close(fd);
}
```

The routine closes the given file descriptor. This causes the protocol driver to free any resources associated with that stream. For example, the driver can unbind the protocol address that had previously been bound to that stream, thereby freeing that address for use by another service user.

Sending Data to the Service Provider

The third routine, inter_snd, passes data to the service provider for transmission to the user at the address specified in addr. The data to be transmitted is contained in the buffer pointed to by buf and contains len bytes. On successful completion, this routine returns the number of bytes of data passed to the service provider; on failure, it returns −1.

EXAMPLE 3-3   inter_snd

```
inter_snd(int fd, char *buf, int len, long *addr)
{
  struct strbuf ctlbuf;
  struct strbuf databuf;
  struct unitdata_req unitdata_req;

  unitdata_req.PRIM_type = UNITDATA_REQ;
  unitdata_req.DEST_addr = addr;

  ctlbuf.len = sizeof(struct unitdata_req);
  ctlbuf.buf = (char *)&unitdata_req;
  databuf.len = len;
  databuf.buf = buf;

  if (putmsg(fd, &ctlbuf, &databuf, 0) < 0)
   return(-1);
   return(len);
}
```

In this example, the data request primitive is packaged with both a control part and a data part. The control part contains a unitdata_req structure that identifies the primitive type and the destination address of the data. The data to be transmitted is placed in the data part of the request message.

Unlike the bind request, the data request primitive requires no acknowledgement from the service provider. In the example, this choice was made to minimize the overhead during data transfer. If the putmsg call succeeds, this routine returns the number of bytes passed to the service provider.

Receiving Data

The final routine in Example 3-4, `inter_rcv`, retrieves the next available data. `buf` points to a buffer where the data should be stored, `len` indicates the size of that buffer, and `addr` points to a long integer where the source address of the data is placed. On successful completion, `inter_rcv` returns the number of bytes of retrieved data; on failure, it returns -1 and an appropriate UNIX system error value.

EXAMPLE 3-4   Receiving Data

```
int inter_rcv (int fd, char *buf, int len, long *addr, int *errorp)
{
  struct strbuf ctlbuf;
  struct strbuf databuf;
  struct unitdata_ind unitdata_ind;
  int retval;
  int flagsp;

 ctlbuf.maxlen = sizeof(struct unitdata_ind);
 ctlbuf.len = 0;
 ctlbuf.buf = (char *)&unitdata_ind;
 databuf.maxlen = len;
 databuf.len = 0;
 databuf.buf = buf;
 flagsp = 0;

 if((retval=getmsg(fd,&ctlbuf,&databuf,&flagsp))<0) {
   *errorp = EIO;
   return(-1);
 }
 if (retval) {
   *errorp = EIO;
   return(-1)
 }
 if (unitdata_ind.PRIM_type != UNITDATA_IND) {
   *errorp = EPROTO;
   return(-1);
 }
 *addr = unitdata_ind.SRC_addr;
 return(databuf.len);
}
```

`getmsg` is called to retrieve the data indication primitive, where that primitive contains both a control and data part. The control part consists of a `unitdata_ind` structure that identifies the primitive type and the source address of the data sender. The data part contains the data itself. In `ctlbuf`, `buf` points to a buffer containing the control information, and `maxlen` indicates the maximum size of the buffer. Similar initialization is done for `databuf`.

The integer pointed to by `flagsp` in the `getmsg` call is set to zero, indicating that the next message should be retrieved from the stream head regardless of its priority. Data arrives in

normal priority messages. If there is no message at the stream head, `getmsg` blocks until a message arrives.

The user's control and data buffers should be large enough to hold any incoming data. If both buffers are large enough, `getmsg` processes the data indication and returns 0, indicating that a full message was retrieved successfully. However, if neither buffer is large enough, `getmsg` only returns the part of the message that fits into each user buffer. The remainder of the message is saved for subsequent retrieval (in message non-discard mode), and a positive, nonzero value is returned to the user. A return value of `MORECTL` indicates that more control information is waiting for retrieval. A return value of `MOREDATA` indicates that more data is waiting for retrieval. A return value of (`MORECTL | MOREDATA`) indicates that data from both parts of the message remain. In the example, if the user buffers are not large enough (that is, `getmsg` returns a positive, nonzero value), the function sets `errno` to `EIO` and fails.

The type of the primitive returned by `getmsg` is checked to make sure it is a data indication (`UNITDATA_IND` in the example). The source address is then set and the number of bytes of data is returned.

The example presented is a simplified service interface. It shows typical uses of `putmsg`(2) and `getmsg`(2). The state transition rules for the interface are not presented and this example does not handle expedited data.

Input and Output Polling

This section describes the synchronous polling mechanism and asynchronous event notification in STREAMS.

User processes can efficiently monitor and control multiple streams with two system calls: `poll`(2) and the `I_SETSIG` `ioctl`(2) command. These calls enable a user process to detect events that occur at the stream head on one or more streams, including receipt of data or messages on the read queue and cessation of flow control on the write queue. Note that `poll`(2) is usable on any character device file descriptor, not just STREAMS.

To monitor streams with `poll`(2), a user process issues that system call and specifies the streams and other files to be monitored, the events to check, and the amount of time to wait for an event. `poll`(2) blocks the process until the time expires or until an event occurs. If an event occurs, it returns the type of event and the descriptor on which the event occurred.

Instead of waiting for an event to occur, a user process can monitor one or more streams while processing other data. To do so, issue the `I_SETSIG` `ioctl`(2), specifying a stream and events (as with `poll`(2)). This `ioctl`(2) does not block the process and force the user process to wait for the event, but returns immediately and issues a signal when an event occurs. The process calls one of `sigaction`(2), `signal`(3c), or `sigset`(3C) to catch the resulting `SIGPOLL` signal.

If any selected event occurs on any of the selected streams, STREAMS sends SIGPOLL to all associated requesting processes. The processes have no information on what event occurred on what stream. A signaled process can get more information by calling poll(2).

Synchronous Input and Output

poll(2) provides a mechanism to identify the streams over which a user can send or receive data. For each stream of interest, users can specify one or more events about which they should be notified. The types of events that can be polled are POLLIN, POLLRDNORM, POLLRDBAND, POLLPRI, POLLOUT, POLLWRNORM, POLLWRBAND, which are detailed in Table 3-3.

TABLE 3-3   Events That Can Be Polled

Event	Description
POLLIN	A message other than high-priority data can be read without blocking. This event is maintained for compatibility with the previous releases of the Oracle Solaris operating environment.
POLLRDNORM	A normal (nonpriority) message is at the front of the stream head read queue.
POLLRDBAND	A priority message (band > 0) is at the front of the stream head queue.
POLLPRI	A high-priority message is at the front of the stream head read queue.
POLLOUT	The normal priority band of the queue is writable (not flow controlled).
POLLWRNORM	The same as POLLOUT.
POLLWRBAND	A priority band greater than 0 of a queue downstream.

Some of the events may not be applicable to all file types. For example, the POLLPRI event usually is not generated when polling a non-STREAMS character device. POLLIN, POLLRDNORM, POLLRDBAND, and POLLPRI are set even if the message is of zero length.

poll(2) checks each file descriptor for the requested events and, on return, indicates which events have occurred for each file descriptor. If no event has occurred on any polled file descriptor, poll(2) blocks until a requested event or timeout occurs. poll(2) takes the following arguments:

- An array of file descriptors and events to be polled.
- The number of file descriptors to be polled.
- The number of milliseconds poll should wait for an event if no events are pending (-1 specifies wait forever).

Example 3-5 shows the use of poll(2). Two separate minor devices of the communications driver are opened, thereby establishing two separate streams to the driver. The pollfd entry

is initialized for each device. Each stream is polled for incoming data. If data arrive on either stream, data is read and then written back to the other stream.

EXAMPLE 3-5 Polling

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

#define NPOLL 2  /* number of file descriptors to poll */
int
main()
{
  struct pollfd pollfds[NPOLL];
  char buf[1024];
  int count, i;

  if ((pollfds[0].fd = open("/dev/ttya", O_RDWR|O_NONBLOCK)) < 0) {
    perror("open failed for /dev/ttya");
    exit(1);
  }
  if ((pollfds[1].fd = open("/dev/ttyb", O_RDWR|O_NONBLOCK)) < 0) {
    perror("open failed for /dev/ttyb");
    exit(2);
  }
```

The variable `pollfds` is declared as an array of the `pollfd` structure, defined in `<poll.h>`, and has the format:

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

For each entry in the array, `fd` specifies the file descriptor to be polled and `events` is a bitmask that contains the bitwise inclusive OR of events to be polled on that file descriptor. On return, the `revents` bitmask indicates which of the requested events has occurred.

The example continues to process incoming data, as shown below:

```
pollfds[0].events = POLLIN; /* set events to poll */
pollfds[1].events = POLLIN; /* for incoming data */
while (1) {
 /* poll and use -1 timeout (infinite) */
 if (poll(pollfds, NPOLL, -1) < 0) {
  perror("poll failed");
  exit(3);
 }
 for (i = 0; i < NPOLL; i++) {
  switch (pollfds[i].revents) {
   default:        /* default error case */
    fprintf(stderr,"error event\n");
    exit(4);
```

```
   case 0:      /* no events */
    break;

   case POLLIN:
    /*echo incoming data on "other" Stream*/
    while ((count = read(pollfds[i].fd, buf, 1024)) > 0)
     /*
      * write loses data if flow control
      * prevents the transmit at this time
      */
      if (write(pollfds[(i+1) % NPOLL].fd buf,
         count) != count)
        fprintf(stderr,"writer lost data");
      break;
   }
 }
}
```

The user specifies the polled events by setting the events field of the pollfd structure to POLLIN. This request tells poll(2) to notify the user of any incoming data on each stream. The bulk of the example is an infinite loop, where each iteration polls both streams for incoming data.

The second argument of poll(2) specifies the number of entries in the pollfds array (2 in this example). The third argument indicates the number of milliseconds poll(2) waits for an event if none has occurred. On a system where millisecond accuracy is not available, timeout is rounded up to the nearest value available on that system. If the value of timeout is 0, poll(2) returns immediately. Here, timeout is set to -1, specifying that poll(2) blocks until a requested event occurs or until the call is interrupted.

If poll(2) succeeds, the program checks each entry in the pollfds array. If revents is set to 0, no event has occurred on that file descriptor. If revents is set to POLLIN, incoming data is available, so all available data is read from the polled minor device and written to the other minor device.

If revents is set to a value other than 0 or POLLIN, an error event must have occurred on that stream because POLLIN was the only requested event. Table 3-4 shows poll error events.

TABLE 3-4   poll Error Events

Error	Description
POLLERR	A fatal error has occurred in a module or driver on the stream associated with the specified file descriptor. Further system calls fail.
POLLHUP	A hangup condition exists on the stream associated with the specified file descriptor. This event and POLLOUT are mutually exclusive; a stream is not writable if a hangup has occurred.
POLLNVAL	The specified file descriptor is not associated with an open stream.

These events cannot be polled for by the user but are reported in `revents` when they occur. They are only valid in the `revents` bitmask.

The example attempts to process incoming data as quickly as possible. However, when writing data to a stream, `write`(2) can block if the stream is exerting flow control. To prevent the process from blocking, the minor devices of the communications driver are opened with the `O_NDELAY` (or `O_NONBLOCK`) flag set, see note. `write`(2) cannot send all the data if flow control is on and `O_NDELAY` (`O_NONBLOCK`) is set. This can happen if the communications driver processes characters slower than the user transmits. If the stream becomes full, the number of bytes `write`(2) sends is less than the requested `count`. For simplicity, the example ignores the data if the stream becomes full, and a warning is printed to `stderr`.

Note - To conform with the IEEE operating system interface standard, POSIX, new applications should use the `O_NONBLOCK` flag. Its behavior is the same as that of `O_NDELAY` unless otherwise noted.

This program continues until an error occurs on a stream, or until the process is interrupted.

Asynchronous Input and Output

`poll`(2) enables a user to monitor multiple streams synchronously. `poll`(2) normally blocks until an event occurs on any of the polled file descriptors. In some applications, however, you want to process incoming data asynchronously. For example, an application can attempt to do some local processing and be interrupted when a pending event occurs. Some time-critical applications must not block, and must have immediate success or failure indication.

The `I_SETSIG` `ioctl`(2) (see `streamio`(7I)) is used to request that a `SIGPOLL` signal be sent to a user process when a specific event occurs. Table 3-5 lists events for `I_SETSIG`. These are similar to those described for `poll`(2).

TABLE 3-5   `I_SETSIG` `ioctl`(2) Events

Event	Description
S_INPUT	A message other than a high-priority message has arrived on a stream head read queue. This event is maintained for compatibility with the previous releases of the Oracle Solaris operating environment.
S_RDNORM	A normal (nonpriority) message has arrived on the stream head read queue.
S_RDBAND	A priority message (band > 0) has arrived on the stream head read queue.
S_HIPRI	A high-priority message has arrived on the stream head read queue.

Event	Description
S_OUTPUT	A write queue for normal data (priority band = 0) is no longer full (not flow controlled). This notifies a user that there is space on the queue for sending or writing normal data downstream.
S_WRNORM	The same as S_OUTPUT.
S_WRBAND	A priority band greater than 0 of a queue downstream exists and is writable. This notifies a user that there is space on the queue for sending or writing priority data downstream.
S_MSG	A signal message sent from a module or driver has arrived on the stream head read queue.
S_ERROR	An error message reaches the stream head.
S_HANGUP	A hangup message sent from a module or driver has arrived at the stream head.
S_BANDURG	When used with S_RDBAND, SIGURG is generated instead of SIGPOLL when a priority message reaches the front of the stream head read queue.

S_INPUT, S_RDNORM, S_RDBAND, and S_HIPRI are set even if the message is of zero length. A user process can handle only high-priority messages by setting the arg to S_HIPRI.

signal Message

STREAMS enables modules and drivers to send a signal to user processes through a special signal message. If the signal specified by the module or driver is not SIGPOLL (see signal(3C)), the signal is sent to the process group associated with the stream. If the signal is SIGPOLL, the signal is only sent to processes that have registered for the signal by using the I_SETSIG ioctl(2).

Extended Signals

So that a process can obtain the band and event associated with SIGPOLL more readily, STREAMS supports extended signals. For the given events, a special code is defined in <sys/siginfo.h> that describes the reason SIGPOLL was generated. Table 3-6 describes the data available in the siginfo_t structure passed to the signal handler.

TABLE 3-6   Data in siginfo_t Structure

Event	si_signo	si_code	si_band	si_errno
S_INPUT	SIGPOLL	POLL_IN	Band readable	Unused
S_OUTPUT	SIGPOLL	POLL_OUT	Band writable	Unused

Event	si_signo	si_code	si_band	si_errno
S_MSG	SIGPOLL	POLL_MSG	Band signaled	Unused
S_ERROR	SIGPOLL	POLL_ERR	Unused	stream error
S_HANGUP	SIGPOLL	POLL_HUP	Unused	Unused
S_HIPRI	SIGPOLL	POLL_PRI	Unused	Unused

Stream as a Controlling Terminal

The controlling terminal can receive signals and send signals. If a foreground process group has the stream as a controlling terminal stream, drivers and modules can use M_SIG messages to send signals to processes.

Job Control

An overview of Job Control is provided here because it interacts with the STREAMS-based terminal subsystem. You can obtain more information on Job Control from the following manual pages: exit(2), getpgid(2), getpgrp(2), getsid(2), kill(2), setpgid(2), setpgrp(2), setsid(2), sigaction(2), signal(3C), sigsend(2), termios(3C), waitid(2), and termio(7I).

Job Control breaks a login session into smaller units called jobs. Each job consists of one or more related and cooperating processes. The *foreground* job, is given complete access to the controlling terminal. The other *background* jobs are denied read access to the controlling terminal and given conditional write and ioctl(2) access to it. The user can stop the executing job and resume the stopped job either in the foreground or in the background.

Under Job Control, background jobs do not receive events generated by the terminal and are not informed with a hangup indication when the controlling process exits. Background jobs that linger after the login session has been dissolved are prevented from further access to the controlling terminal, and do not interfere with the creation of new login sessions.

The following list defines terms associated with Job Control:

Background process group	A process group that is a member of a session that established a connection with a controlling terminal and is not the foreground process group.
Controlling process	A session leader that established a connection to a controlling terminal.

Controlling terminal	A terminal that is associated with a session. Each session can have at most one controlling terminal associated with it, and a controlling terminal can be associated with at most one session. Certain input sequences from the controlling terminal cause signals to be sent to the process groups in the session associated with the controlling terminal.
Foreground process group	Each session that establishes a connection with a controlling terminal distinguishes one process group of the session as a foreground process group. The foreground process group has certain privileges that are denied to background process groups when accessing its controlling terminal.
Orphaned process group	A process group in which the parent of every member in the group is either a member of the group, or is not a member of the process group's session.
Process group	Each process in the system is a member of a process group that is identified by a process group ID. Any process that is not a process group leader can create a new process group and become its leader. Any process that is not a process group leader can join an existing process group that shares the same session as the process. A newly created process joins the process group of its creator.
Process group leader	A process whose process ID is the same as its process group ID.
Process group lifetime	A time period that begins when a process group is created by its process group leader and ends when the last process that is a member in the group leaves the group.
Process ID	A positive integer that uniquely identifies each process in the system. A process ID cannot be reused by the system until the process lifetime, process group lifetime, and session lifetime end for any process ID, process group ID, and session ID sharing that value.
Process lifetime	A period that begins when the process is forked and ends after the process exits, when its termination has been acknowledged by its parent process.
Session	Each process group is a member of a session that is identified by a session ID.
Session ID	A positive integer that uniquely identifies each session in the system. It is the same as the process ID of its session leader (POSIX).
Session leader	A process whose session ID is the same as its process and process group ID.

Session lifetime   A period that begins when the session is created by its session leader and ends when the lifetime of the last process group that is a member of the session ends.

The following signals manage Job Control: (see also `signal`(3C)) :

SIGCONT   Sent to a stopped process to continue it.

SIGSTOP   Sent to a process to stop it. This signal cannot be caught or ignored.

SIGTSTP   Sent to a process to stop it. It is typically used when a user requests to stop the foreground process.

SIGTTIN   Sent to a background process to stop it when it attempts to read from the controlling terminal.

SIGTTOU   Sent to a background process to stop it when a user attempts to write to or modify the controlling terminal.

A session can be allocated a controlling terminal. For every allocated controlling terminal, Job Control elevates one process group in the controlling process's session to the status of foreground process group. The remaining process groups in the controlling process's session are background process groups. A controlling terminal gives a user the ability to control execution of jobs within the session. Controlling terminals are critical in Job Control. A user can cause the foreground job to stop by typing a predefined key on the controlling terminal. A user can inhibit access to the controlling terminal by background jobs. Background jobs that attempt to access a terminal that has been so restricted is sent a signal that typically causes the job to stop. (See .)

Job Control requires support from a line-discipline module on the controlling terminal's stream. The `TCSETA`, `TCSETAW`, and `TCSETAF` commands of `termio`(7I) allow a process to set the following line discipline values relevant to Job Control:

SUSP character   A user-defined character that, when typed, causes the line discipline module to request that the stream head send a `SIGTSTP` signal to the foreground process, which by default stops the members of that group. If the value of `SUSP` is zero, the `SIGTSTP` signal is not sent, and the `SUSP` character is disabled.

TOSTOP flag   If `TOSTOP` is set, background processes are inhibited from writing to their controlling terminal. A line discipline module must record the `SUSP` suspend character and notify the stream head when the user has typed it, and record the state of the `TOSTOP` bit and notify the stream head when the user has changed it.

Allocation and Deallocation of Streams

A stream is allocated as a controlling terminal for a session if it:

- Is acting as a terminal.
- Is not already allocated as a controlling terminal.
- Is opened by a session leader that does not have a controlling terminal.

  Controlling terminals are allocated with open(2). The device must inform the stream head that it is acting as a terminal.

Hungup Streams

When a stream head receives a hangup message from a device or module, it is marked as hung up. A stream that is marked as hung up is allowed to be reopened by its session leader if it is allocated as a controlling terminal, and by any process if it is not allocated as a controlling terminal. This way, the hangup error can be cleared without forcing all file descriptors to be closed first.

If the reopen is successful, the hangup condition is cleared.

Hangup Signals

When the SIGHUP signal is generated by a hangup message instead of a signal message, the signal is sent to the controlling process instead of the foreground process group. The allocation and deallocation of controlling terminals to a session is the responsibility of that process group.

Accessing the Controlling Terminal

If a process attempts to access its controlling terminal after it has been deallocated, access is denied. If the process is not holding or ignoring SIGHUP, it is sent a SIGHUP signal. Otherwise, the access fails with an EIO error.

Members of background process groups have limited access to their controlling terminals:

- If the background process is ignoring or holding the SIGTTIN signal or is a member of an orphaned process group, an attempt to read from the controlling terminal fails with an EIO error. Otherwise, the process is sent a SIGTTIN signal, which by default stops the process.
- If the process is attempting to write to the terminal and if the terminal's TOSTOP flag is clear, the process is allowed access.

- If the terminal's TOSTOP flag is set and a background process is attempting to write to the terminal, the write succeeds if the process is ignoring or holding SIGTTOU. Otherwise, the process stops except when it is a member of an orphaned process group, in which case it is denied access to the terminal and it is returned an EIO error.

  If a background process is attempting to perform a destructive ioctl(2) (one that modifies terminal parameters), the ioctl(2) call succeeds if the process is ignoring or holding SIGTTOU. Otherwise, the process stops except when the process is a member of the orphaned process group. In that case the access to the terminal is denied and an EIO error is returned.

♦ ♦ ♦ **C H A P T E R 4**

Application Access to the STREAMS Driver and Module Interfaces

This chapter describes getting messages into and out of the driver from an application level. It shows the relationship between messages overall and the specific ioctl(2) calls that pertain to application-level operations.

- "System Calls Used" on page 57
- "Module and Driver ioctl Calls" on page 58
- "Flush Handling" on page 65

System Calls Used

Table 4-1 summarizes the system calls commonly used in controlling and transferring data and messages within a stream.

TABLE 4-1   System Calls Used

System Call	Description
read(2)	Reads data from a stream
write(2)	Writes data to a stream
ioctl(2)	Controls a stream
getmsg(2)	Receives a message at the stream head
getpmsg(2)	Receives a priority message at the stream head
putmsg(2)	Sends a message downstream
putpmsg(2)	Sends a priority message downstream
poll(2)	Identifies files on which a user can send or receive messages, or on which certain events have occurred (historically, it was unnecessarily restricted to streams)

System Call	Description
pipe(2)	Creates a bidirectional channel that provides a communication path between multiple processes

Module and Driver `ioctl` Calls

STREAMS is a special type of character device driver that is different from the historical character input/output (I/O) mechanism. In this section, the phrases *character I/O mechanism* and *I/O mechanism* refer only to that part of the mechanism that existed before STREAMS.

The character I/O mechanism handles all ioctl(2) calls transparently. That is, the kernel expects all ioctl(2) to be handled by the device driver associated with the character special file on which the call is sent. All ioctl(2) calls are sent to the driver, which is expected to perform all validation and processing other than file descriptor validity checking. The operation of any specific ioctl(2) is dependent on the device driver. If the driver requires data to be transferred in from user space, it will use the kernel `ddi_copyin` function. It may also use `ddi_copyout` to transfer any data results to user space.

With STREAMS, there are a number of differences from the character I/O mechanism that impart ioctl(2) processing.

First, there is a set of generic STREAMS ioctl(2) command values recognized and processed by the stream head. This is described in streamio(7I). The operation of the generic STREAMS ioctl(2) is generally independent of the presence of any specific module or driver on the stream.

The second difference is the absence of user context in a module and driver when the information associated with the ioctl(2) is received. This prevents use of ddi_copyin(9F) or ddi_copyout(9F) by the module. This also prevents the module and driver from associating any kernel data with the currently running process. (By the time the module or driver receives the ioctl(2), the process generating it probably will no longer be running.)

A third difference is that for the character I/O mechanism, all ioctl(2) are handled by the single driver associated with the file. In STREAMS, there can be multiple modules on a stream and each one can have its own set of ioctl(2) calls. That is, the ioctl(2) that can be used on a stream can change as modules are pushed and popped.

STREAMS provides the capability for user processes to perform control functions on specific modules and drivers in a stream with ioctl(2) calls. Most streamio(7I) ioctl(2) commands go no further than the stream head. They are fully processed there and no related

messages are sent downstream. However, certain commands and all unrecognized commands cause the stream head to create an `M_IOCTL </function>` message, which includes the `ioctl(2)` arguments, and send the message downstream to be received and processed by a specific module or driver. The `M_IOCTL </function>` message is the initial message type that carries `ioctl(2)` information to modules. Other message types are used to complete the `ioctl(2)` processing in the stream. In general, each module must uniquely recognize and act on specific `M_IOCTL </function>` messages.

STREAMS `ioctl(2)` handling is equivalent to the transparent processing of the character I/O mechanism. STREAMS modules and drivers can process `ioctl(2)` generated by applications that are implemented for a non-STREAMS environment.

General `ioctl` Processing

STREAMS blocks a user process that issues an `ioctl(2)` and causes the stream head to generate an `M_IOCTL </function>` message. The process remains blocked until one of the following occurs:

- A module or a driver responds with an `M_IOCACK` (`ack`, positive acknowledgement) message or an `M_IOCNAK` (`nak`, negative acknowledgement) message.
- No message is received and the request times out.
- The `ioctl(2)` is interrupted by the user process.
- An error condition occurs. For the `I_STR` `ioctl(2)`, the timeout period can be a user-specified interval or a default. For the other `ioctl(2)` calls, the default value (infinite) is used.

For an `I_STR`, the STREAMS module or driver that generates a positive acknowledgement message can also return data to the process in the message. An alternate means to return data is provided with transparent `ioctl(2)`. If the stream head does not receive a positive or negative acknowledgement message in the specified time, the `ioctl(2)` call fails.

A module that receives an unrecognized `M_IOCTL` message must pass it on unchanged. A driver that receives an unrecognized `M_IOCTL` must produce a negative acknowledgement.

The two STREAMS `ioctl(2)` mechanisms, `I_STR` and transparent, are described next. (Here, `I_STR` means the `streamio(7I)` `I_STR` command and implies the related STREAMS processing unless noted otherwise.) `I_STR` has a restricted format and restricted addressing for transferring `ioctl(2)`-related data between user and kernel space. It requires only a single pair of messages to complete `ioctl(2)` processing. The transparent mechanism is more general and has almost no restrictions on `ioctl(2)` data format and addressing. The transparent mechanism

generally requires that multiple pairs of messages be exchanged between the stream head and module to complete the processing.

This is a rather simplistic view. Nothing prevents a given ioctl(2) from being issued either directly (transparent) or by means of `I_STR`. Furthermore, ioctl(2) calls issued through `I_STR` potentially can require further processing of the form typically associated with transparent ioctl(2).

`I_STR ioctl` Processing

The `I_STR` ioctl(2) provides a capability for user applications to perform module and driver control functions on STREAMS files. `I_STR` allows an application to specify the ioctl(2) timeout. It encourages that all user ioctl(2) data (to be received by the destination module) be placed in a single block that is pointed to from the user `strioctl` structure. The module can also return data to this block.

Transparent `ioctl` Processing

The transparent STREAMS ioctl(2) mechanism enables application programs to perform module and driver control functions with ioctl(2) calls other than `I_STR`. It transparently supports applications developed prior to the introduction of STREAMS. It alleviates the need to recode and recompile the user-level software to run over STREAMS files. More importantly, applications do not have to package their ioctl(2) requests into the form demanded by `I_STR`.

The mechanism extends the data transfer capability for STREAMS ioctl(2) calls beyond those provided in the `I_STR` form. Modules and drivers can transfer data between their kernel space and user space in any ioctl(2) that has a value of the command argument not defined in streamio(7I). These ioctl(2) are known as transparent ioctl(2) to differentiate them from the `I_STR` form. Existing applications that use non-STREAMS character devices require transparent processing for ioctl(2) if the device driver is converted to STREAMS. The ioctl(2) data can be in any format mutually understood by the user application and module.

The transparent mechanism also supports STREAMS applications that send ioctl(2) data to a driver or module in a single call, where the data may not be in a form readily embedded in a single user block. For example, the data may be contained in nested structures and different user space buffers.

I_LIST ioctl

The I_LIST ioctl(2) supports the strconf(1) and strchg(1) commands that are used to query or change the configuration of a stream. Only the root user or an owner of a STREAMS device can alter the configuration of that stream.

strchg(1) does the following:

- Pushes one or more modules on the stream.
- Pops the topmost module off the stream.
- Pops all the modules off the stream.
- Pops all modules up to but not including a specified module.

strconf(1) does the following:

- Checks if the specified module is present on the stream.
- Prints the topmost module of the stream.
- Prints a list of all modules and the topmost driver on the stream. If the stream contains a multiplexing driver, the strchg and strconf commands will not recognize any modules below that driver.

The I_LIST ioctl(2), illustrated in Example 4-1 , performs two functions. When the third argument of the ioctl(2) call is NULL,

```
if ((mods = ioctl(s, I_LIST, 0)) < 0) {
```

the return value of the call indicates the number of modules, plus the driver, present on the stream. For example, if there are two modules above the driver, 3 is returned. On failure, errno may be set to a value specified in streamio(7I). The second function of the I_LIST ioctl(2) is to copy the module names found on the stream to the user-supplied buffer. The address of the buffer in user space and the size of the buffer are passed to the ioctl(2) through a structure str_mlist that is defined as:

```
struct str_mlist {
  char l_name[FMNAMESZ+1]; /* space for holding a module name*/
};

 struct str_list {
  int sl_nmods;  /* #of modules for which space is allocated */
  struct str_mlist *sl_modlist; /*addr of buf for names*/
};
```

Here sl_nmods is the number of modules in the sl_modlist array that the user has allocated. Each element in the array must be at least FMNAMESZ+1 bytes long. The array is FMNAMESZ+1 so the extra byte can hold the NULL character at the end of the string. FMNAMESZ is defined by <sys/conf.h>.

The amount of space to allocate for module names is indicated by the number of modules in the STREAM. This is not completely reliable because another module might be pushed onto the stream after the application invokes the I_LIST ioctl(2) with the NULL argument and before it invokes the I_LIST ioctl(2) with the structure argument.

The I_LIST call with `arg` pointing to the `str_list` structure returns the number of entries that have been filled into the `sl_modlist` array (the number represents the number of modules including the driver). If there is not enough space in the `sl_modlist` array or `sl_nmods` is less than 1, the I_LIST call fails and `errno` is set to EINVAL. If `arg` or the `sl_modlist` array points outside the allocated address space, EFAULT is returned.

EXAMPLE 4-1   I_LIST ioctl

```
#include <stdio.h>
#include <string.h>
#include <stropts.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/socket.h>

main(int argc, const char **argv)
{
  int s, i;
  unsigned int mods;
  struct str_list mod_list;struct str_mlist *mlist;
  /* Get a socket... */

  if ((s = socket(AF_INET, SOCK_STREAM, 0)) <= 0) {
   perror("socket: ");
   exit(1);
  }

  /* Determine the number of modules in the stream */
  if ((mods = ioctl(s, I_LIST, 0)) < 0) {
   perror("I_LIST ioctl");
  }
  if (mods == 0) {
   printf("No modules\n");
   exit(1);
  } else {
   printf("%d modules\n", mods);
  }

  /* Allocate memory for all of the module names */
  mlist = (struct str_mlist *)
  calloc(mods, sizeof (struct str_mlist));
  if (mlist == 0) {
   perror("malloc failure");
   exit(1);
  }
  mod_list.sl_modlist = mlist;
```

```
 mod_list.sl_nmods = mods;
 /* Do the ioctl and get the module names... */
 if (ioctl(s, I_LIST, &mod_list) < 0) {
  exit(1);
 }

 /* Print out the name of the modules... */
 for (i = 0; i < mods; i++) {
  printf("s: %s\n", mod_list.sl_modlist[i].l_name);
 }

 /* Free the calloc'd structures... */
 free(mlist);
 return(0);
}
```

Other `ioctl` Commands

`streamio`(7I) details the `ioctl`(2) commands shown in Table 4-2.

TABLE 4-2   Other `ioctl` Commands

`ioctl` Calls	Description
I_ANCHOR	Prevents the removal of a STREAMS module with an `I_POP` call. Any process can place an anchor on a stream, but once placed, an anchor can only be removed by a privileged process.
I_LOOK	Retrieves the name of the module just below the stream head
I_FLUSH	Flushes all input or output queues
I_FLUSHBAND	Flushes a band of messages
I_FIND	Compares the names of all modules present in the stream to a specific name
I_PEEK	Enables the user to look at information in the first message on the stream head read queue without taking the message off the queue
I_SRDOPT	Sets the read mode using a series of flag options in the argument
I_GRDOPT	Indicates the read mode in an `int`
I_NREAD	Counts the data bytes in data blocks in the first message on the stream head read queue
I_FDINSERT	Creates a message from a user buffer, adds information about another stream, and sends the message downstream
I_SWROPT	Sets the write mode using the value of the argument
I_GWROPT	Returns the current write mode setting
I_SENDFD	Requests that the stream send a message with file pointer to the stream head at the other end of a STREAMS pipe

`ioctl` Calls	Description
`I_RECVFD`	Retrieves the file descriptor of the message sent by an `I_SENDFD` ioctl(2) over a STREAMS pipe
`I_ATMARK`	Lets the application see if the current message on the stream head read queue is marked by a module downstream
`I_CKBAND`	Checks if the message of a given priority band exists on the stream head read queue
`I_GETBAND`	Returns the priority band of the first message on the stream head read queue
`I_CANPUT`	Checks if a certain band is writable
`I_SETCLTIME`	Enables the user to set the time the stream head will delay when a stream is closing if data exists on the write queues
`I_GETCLTIME`	Returns the close time delay
`I_LINK`	Connects two streams
`I_UNLINK`	Disconnects two streams
`I_PLINK`	Connects two streams with a persistent link below a multiplexing driver
`I_PUNLINK`	Disconnects two streams that have been connected with a persistent link

Message Direction

Various system calls let the user create messages and send them downstream and prioritize the messages.

TABLE 4-3   Send and Receive Messages

putmsg(2)	Creates a message from the caller supplied control and data buffers and sends the message downstream
putpmsg(2)	Does the same as putmsg(2) and enables the caller to specify a priority band for the message
getmsg(2)	Retrieves `M_DATA`, `M_PROTO`, or `M_PCPROTO` or high priority messages from the stream head, and places the contents into two user buffers
getpmsg(2)	Does the same as `getmsg(2)` and enables the caller to specify a priority band for the message

The stream head guarantees that the control part of a message generated by putmsg(2) is at least 64 bytes long. This promotes reusability of the buffer. When the buffer is a reasonable size, modules and drivers may reuse the buffer for other headers.

`stropts.h` contains the specification of `strbuf`, which describes the control and data buffers.

Flush Handling

All modules and drivers are expected to handle the flushing of messages. The user may cause data to be flushed of queued messages from a stream by the submission of an `I_FLUSH` `ioctl(2)`. Data may be flushed from the read side, write side, or both sides of a stream.

```
ioctl (fd, I_FLUSH, arg);
```

Table 4-4 describes the arguments that may be passed to `M_FLUSH`.

TABLE 4-4   `M_FLUSH` Arguments and `bi_flag` values

Flag	Description
FLUSHR	Flushes read side of stream
FLUSHW	Flushes write queue
FLUSHRW	Flushes both read and write queues

In addition to being able to flush all the data from a queue, a specific band may be flushed using the `I_FLUSHBAND` `ioctl(2)`.

```
ioctl (fd, I_FLUSHBAND, bandp);
```

The `ioctl(2)` is passed a pointer to a `bandinfo` structure. The `bi_pri` field indicates the band priority to be flushed (values from 0 to 255). The `bi_flag` field indicates the type of flush to do. The legal values for `bi_flag` are defined in Table 4-4. `bandinfo` has the following format:

```
struct bandinfo {
  unsigned char    bi_pri;
  int              bi_flag;
};
```

See "Flushing Priority Band" on page 149, which describes `M_FLUSHBAND` processing, for details on how modules and drivers should handle flush band requests.

♦ ♦ ♦ **C H A P T E R 5**

5

STREAMS Administration

This chapter describes the tools available to administer STREAMS. It shows how to keep track of names, where to find modules, and how to monitor statistics. Kernel debugging is covered in Chapter 14, "Debugging STREAMS-based Applications".

- "Administration Tools" on page 67
- "Autopush Facility" on page 68
- "Administration Tool Description" on page 70

Administration Tools

Table 5-1 identifies some common tools available for monitoring, logging, and administering STREAMS.

TABLE 5-1 Tools Available for STREAMS Administration

Tool	Description
autopush(1M)	Configures list of modules to be automatically pushed
crash	Examines system memory images **Note -** crash has reached EOL and is not supported in the Solaris 9 operating environment. For information about how to transition from crash to mdb, see the *Solaris Modular Debugger Guide*.
fdetach(1M)	Detaches a name from a file descriptor
strchg(1)	Prints STREAMS trace messages
strchg(1), strconf(1)	Changes or queries a stream configuration
strerr(1M)	Logs STREAMS errors
modload(1M)	Loads a kernel module
modunload(1M)	Unloads a kernel module

Tool	Description
modinfo(1M)	Displays information about loaded kernel modules

Autopush Facility

The autopush facility (see autopush(1M)) enables administrators to specify a list of modules to be automatically pushed onto the stream whenever a STREAMS device is opened. A prespecified list (/etc/iu.ap) of modules can be pushed onto the stream if the STREAMS device is not already open.

The STREAMS Administrative Driver (SAD) (see sad(7D)) provides an interface to the autopush mechanism. System administrators can open the SAD and set or get autopush information on other drivers. The SAD caches the list of modules to push for each driver. When the driver is opened, if not already open, the stream head checks the SAD's cache to determine if the device is configured to have modules automatically pushed. If an entry is found, the modules are pushed. If the device is open, another open does not cause the list of the prespecified modules to be pushed again.

The autopush SAD_SAP command ap_cmd field specifies the configuration options:

SAP_ONE   Configure each minor device (that is, a specific major and minor device number).

SAP_RANGE   Configure a range of minor devices within a major device.

SAP_ALL   Configure all minor devices within a major device.

SAP_CLEAR   Undo configuration information for a driver.

In addition, when configuring the module list, an optional anchor can be placed within the list. (See "STREAMS Anchors" on page 222 for more information.)

When the configuration list is cleared, a range of minor devices has to be cleared as a range and not in parts.

Application Interface

The SAD is accessed through /dev/sad/admin or /dev/sad/user. After the device is initialized, a program can be run to perform any needed autopush configuration. The program should open the SAD, read a configuration file to find out what modules must be configured for which devices, format the information into strapush structures, and perform the necessary SAD_SAP ioctl(2)s. See sad(7D) for more information.

All autopush operations are performed through an ioctl(2) command to set or get autopush information. Only the root user can set autopush information, but any user can get the autopush information for a device.

In the ioctl(2) call, the parameters are the file descriptor of the SAD, either SAD_SAP (set autopush information) or SAD_GAP (get autopush information), and a pointer to a strapush structure.

strapush is defined as:

```
/*
 * maximum number of modules that can be pushed on a
 * Stream using the autopush feature should be no greater
 * than nstrpush
 */
#define MAXAPUSH 8

/* autopush information common to user and kernel */

struct apcommon {
  uint apc_cmd;            /* command - see below */
  major_t apc_major;       /* major device number */
  minor_t apc_minor;       /* minor device number */
  minor_t apc_lastminor;   /* last minor dev # for range */
  uint apc_npush;          /* number of modules to push */
};

/* ap_cmd - various options of autopush */
#define SAP_CLEAR    0     /* remove configuration list */
#define SAP_ONE      1     /* configure one minor device*/
#define SAP_RANGE    2     /* config range of minor devices */
#define SAP_ALL      3     /* configure all minor devices */

/* format of autopush ioctls */
struct strapush {
  struct apcommon sap_common;
  char sap_list[MAXAPUSH] [FMNAMESZ + 1]; /* module list */
};

#define sap_cmd             sap_common.apc_cmd
#define sap_major           sap_common.apc_major
#define sap_minor           sap_common.apc_minor
#define sap_lastminor       sap_common.apc_lastminor
#define sap_npush           sap_common.apc_npush
```

A device is identified by its major device number, sap_major. A SAD_CMD ioctl(2) is one of the following commands:

SAP_CLEAR   Clears the previous settings by removing the entry with the matching sap_major and sap_minor fields

SAP_ONE   Configures a single minor device, sap_minor, of a driver

SAP_RANGE   Configures a range of minor devices from `sap_minor` to
                               `sap_lastminor`, inclusive

SAP_ALL   Configures all minor devices of a device

The list of modules is specified as a list of module names in `sap_list`. The maximum number
of modules to push automatically is defined by `MAXAPUSH`.

A user can query the current configuration status of a given major or minor device by issuing
the `SAD_GAP` `ioctl(2)` with `sap_major` and `sap_minor` values of the device set. On successful
return from this system call, the `strapush` structure will be filled in with the corresponding
information for that device. The maximum number of entries the SAD driver can cache is
determined by the tunable parameter `NAUTOPUSH` found in the SAD driver's master file.

The following is an example of an autopush configuration file in `/etc/iu.ap`:

```
# major    minor   lastminor    modules
  wc        0       0            ldterm ttcompat
  zs        0       1            ldterm ttcompat
  ptsl      0       15           ldterm ttcompat
```

The first line is the configuration of a single minor device whose major name is `wc` and minor
numbers start at 0 and end at 0, creating only one minor number. The modules automatically
pushed are `ldterm` and `ttcompat`. The second line represents the configuration of the `zs` driver.
The minor device numbers are 0 and 1. The last line allows minor device numbers from 0 to 15
for the `ptsl` driver.

Administration Tool Description

STREAMS error and trace loggers are provided for debugging and for administering
STREAMS modules and drivers. This facility consists of log(7D), strace(1M),
strclean(1M), strerr(1M), and the strlog(9F) function.

`strace` Command

strace(1M) is a utility that displays the messages in a specified STREAMS log. The log to
display is identified by STREAMS module ID number, a sub-ID number, and the priority level.

`strlog` Command

`strlog`(9F) sends formatted text to `log`(7D) driver. Required definitions are contained in `<sys/strlog.h>` and `<sys/log.h>`. The call specifies the STREAMS module ID number, a sub-ID number, and the priority level. A flag parameter can specify any combination of:

`SL_ERROR`	The message is for the error logger
`SL_TRACE`	The message is for the tracer
`SL_FATAL`	Advisory notification of a fatal error
`SL_NOTIFY`	Modifies the `SL_ERROR` flag to request that a copy of the message be mailed to the system administrator
`SL_CONSOLE`	Log the message to the console
`SL_WARN`	Warning message
`SL_NOTE`	Notice the message

The flags are followed by a `printf`(3C)-style format string, but `%s`, `%e`, `%E`, `%g`, and `%G` conversion specifications are not recognized. Up to `NLOGARGS` of numeric or character arguments can be specified.

`strqget` Command

`strqget`(9F) gets information about a queue or band of a queue. The information is returned in a `long`. The stream must be frozen by the caller when calling `strqget`.

`strqset` Command

`strqset`(9F) changes information about a queue or band of the queue. The updated information is provided in an `int`. If the field is read-only, `EPERM` is returned and the field is left unchanged. See `<sys/stream.h>` for valid values. The stream must be frozen by the caller when calling `strqset`(9F).

`strerr` Daemon

`strerr`(1M) is the STREAMS error logger daemon.

6 ♦♦♦ C H A P T E R 6

Pipes and Queues

This chapter covers communication between processes using STREAMS-based pipes and named pipes. Discussion is limited to communications between applications.

Overview of Pipes and FIFOs

A pipe in the SunOS 5.6 system provides a communication path between multiple processes. Prior to the SunOS 5.0 release, SunOS had *standard* pipes and *named* pipes (also called FIFOs). With standard pipes, one end was opened for reading and the other end for writing, so data flow was unidirectional. FIFOs had only one end and typically one process opened the file for reading and another process opened the file for writing. Data written into the FIFO by the writer could then be read by the reader.

To provide greater support and development flexibility for networked applications, pipes and FIFOs are STREAMS-based in the SunOS 5 software. The interface is unchanged but the underlying implementation is changed. When a pipe is created through the `pipe`(2) interface, two streams are opened and connected. Data flow is serial.

The remainder of this chapter uses the terms *pipe* and *STREAMS-based pipe* interchangeably to mean a STREAMS-based pipe.

Note - After both ends of a FIFO have been opened, there is no guarantee that further calls to open O_RDONLY (O_WRONLY) will synchronize with later calls to open O_WRONLY (O_RDONLY) until both ends of the FIFO have been closed by all readers and writers. Any data written into a FIFO will be lost if both ends of the FIFO are closed before the data is read.

Creating and Opening Pipes and FIFOs

A named pipe, also called a FIFO, is a pipe identified by an entry in a file system's name space. FIFOs are created using mknod(2), mkfifo(1M), or the mknod(1M) command. They are removed using unlink(2) or the rm(1) command.

FIFOs look like regular file system nodes, but are distinguished from them by a p in the first column when the ls -l command is run.

```
 /usr/sbin/mknod xxx pls -l xxx
prw-r--r-- 1 guest other 0 Aug 26 10:55 xxx
echoput> hello.world>xxx &put>
[1] 8733
cat xxx
hello world
[1]  + Done
rm xxx
```

FIFOs are unidirectional: that is, one end of the FIFO is used for writing data, the other for reading data. FIFOs allow simple one-way interprocess communication between unrelated processes. Modules may be pushed onto a FIFO. Data written to the FIFO is passed down the write side of the module and back up the read side as shown in Figure 6-1.

FIGURE 6-1   Pushing Modules on a STREAMS-based FIFO
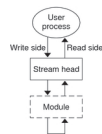
FIFOs are opened in the same manner as other file system nodes with open(2). Any data written to the FIFO can be read from the same file descriptor in the first-in, first-out manner (serial, sequentially). Modules can also be pushed on the FIFO. See open(2) for the restrictions that apply when opening a FIFO. If O_NDELAY or O_NONBLOCK is not specified, an open on a FIFO blocks until both a reader and a writer are present.

Named or mounted streams provide a more powerful interface for interprocess communications than does a FIFO. See "Named Streams" on page 78 for details.

A STREAMS-based pipe, also referred to as an anonymous pipe, is created using pipe(2), which returns two file descriptors, fd[0] and fd[1], each with its own stream head. The ends of the pipe are constructed so that data written to either end of a pipe may be read from the opposite end.

STREAMS modules can be added to a pipe with I_PUSH ioctl(2). A module can be pushed onto one or both ends of the pipe (see Figure 6-2). However, if a module is pushed onto one end of the pipe, that module cannot be popped from the other end.

FIGURE   6-2   Pushing Modules on a STREAMS-based Pipe
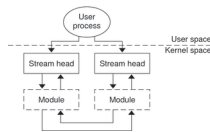
Using Pipes and FIFOs

Pipes and FIFOs can be accessed through the operating system routines read(2), write(2), ioctl(2), close(2), putmsg(2), putpmsg(2), getmsg(2), getpmsg(2), and poll(2). For FIFOs, open(2) is also used.

Reading From a Pipe or FIFO

read(2) or getmsg(2)) are used to read from a pipe or FIFO. Data can be read from either end of a pipe. On success, the read(2) returns the number of bytes read and buffered. When the end of the data is reached, read(2) returns 0.

When a user process attempts to read from an empty pipe (or FIFO), the following happens:

- If one end of the pipe is closed, 0 is returned, indicating the end of the file.
- If the write side of the FIFO has closed, read(2) returns 0 to indicate the end of the file.
- If some process has the FIFO open for writing, or both ends of the pipe are open, and O_NDELAY is set, read(2) returns 0.
- If some process has the FIFO open for writing, or both ends of the pipe are open, and O_NONBLOCK is set, read(2) returns -1 and sets errno to EAGAIN.
- If O_NDELAY and O_NONBLOCK are not set, the read call blocks until data is written to the pipe, until one end of the pipe is closed, or the FIFO is no longer open for writing.

Writing to a Pipe or FIFO

When a user process calls write(2), data is sent down the associated stream. If the pipe or FIFO is empty (no modules pushed), the data that is written is placed on the read queue of the other stream for pipes, and on the read queue of the same stream for FIFOs. Since the size of a pipe is the number of unread data bytes, the written data is reflected in the size of the other end of the pipe.

Zero-Length Writes

If a user process issues write(2) with 0 as the number of bytes to send a pipe or FIFO, 0 is returned, and, by default, no message is sent down the stream. However, if a user must send a zero-length message downstream, SNDZERO ioctl(2) can be used to change this default behavior. If SNDZERO is set in the stream head, write(2) requests of 0 bytes generate a zero-length message and send the message down the stream. If SNDZERO is not set, no message is generated and 0 is returned to the user.

The SNDZERO bit may be changed by the I_SWROPT ioctl(2). If the arg in the ioctl(2) has SNDZERO set, the bit is turned on. If the arg is set to 0, the SNDZERO bit is turned off.

The I_GWROPT ioctl(2) is used to get the current write settings.

Atomic Writes

If multiple processes simultaneously write to the same pipe, data from one process can be interleaved with data from another process, if modules are pushed on the pipe or the write is greater than PIPE_BUF. The order of data that is written is not necessarily the order of data that is read. To ensure that writes of less than PIPE_BUF bytes are not interleaved with data written by other processes, any modules pushed on the pipe should have a maximum packet size of at least PIPE_BUF.

Note - PIPE_BUF is an implementation-specific constant that specifies the maximum number of bytes that are atomic when writing to a pipe. When writing to a pipe, write requests of PIPE_BUF or fewer bytes are not interleaved with data from other processes doing writes to the same pipe. However, write requests of more than PIPE_BUF bytes may have data interleaved on arbitrary byte boundaries with writes by other processes whether or not the O_NONBLOCK or O_NDELAY flag is set.

If the module packet size is at least the size of PIPE_BUF, the stream head packages the data in such a way that the first message is at least PIPE_BUF bytes. The remaining data may be

packaged into smaller or equal-sized blocks depending on buffer availability. If the first module on the stream cannot support a packet of PIPE_BUF, atomic writes on the pipe cannot be guaranteed.

Closing a Pipe or FIFO

close(2) closes a pipe or FIFO and dismantles its associated streams. On the last close of one end of a pipe, an M_HANGUP message is sent to the other end of the pipe. Subsequent read(2) or getmsg(2) calls on that stream head return the number of bytes read and zero when there are no more data. Subsequent write(2) or putmsg(2) requests fail with errno set to EPIPE. If the other end of the pipe is mounted, the last close of the pipe forces it to be unmounted.

Flushing Pipes and FIFOs

When the flush request is initiated from an ioctl(2) or from flushq(9F), the FLUSHR or the FLUSHW bits of an M_FLUSH message must be switched. Bits are switched at the point where the M_FLUSH message is passed from a write queue to a read queue. This point is also known as the *midpoint* of the pipe.

The midpoint of a pipe is not always easily detectable, especially if there are numerous modules pushed on either end of the pipe. In that case, some mechanism needs to intercept all messages passing through the stream. If the message is an M_FLUSH message and it is at the stream midpoint, the flush bits need to be switched.

This bit switching is handled by the pipemod module. pipemod should be pushed onto a pipe or FIFO where flushing of any kind will take place. The pipemod(7M) module can be pushed on either end of the pipe. The only requirement is that it is pushed onto an end that previously did not have modules on it. That is, pipemod(7M) must be the first module pushed onto a pipe so that it is at the midpoint of the pipe itself.

The pipemod(7M) module handles only M_FLUSH messages. All other messages are passed to the next module using the putnext(9F) utility routine. If an M_FLUSH message is passed to pipemod(7M) and the FLUSHR and FLUSHW bits are set, the message is not processed but is passed to the next module using putnext(9F). If only the FLUSHR bit is set, it is turned off and the FLUSHW bit is set. The message is then passed to the next module, using putnext(9F). Similarly, if the FLUSHW bit was the only bit set in the M_FLUSH message, it is turned off and the FLUSHR bit is turned on. The message is then passed to the next module on the stream.

The pipemod(7M) module can be pushed on any stream if it requires the bit switching.

Named Streams

The name of a stream or STREAMS–based pipe often associates the stream with an existing node in the file system name space. This allows unrelated processes to open the pipe and exchange data with the application. The following interfaces support naming a stream or STREAMS–based pipe.

fattach(3C)   Attaches a stream file descriptor to a node in the file system name space, thus naming the stream.

fdetach(3C)   Detaches a named stream file descriptor from its node in the file system name space, thus unnaming the stream.

isastream(3C)   Tests whether a file descriptor is associated with a stream.

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

Unique Connections

With named pipes, client processes may communicate with a server process using the connld module which lets a client process get a unique, non-multiplexed connection to a server. connld(7M) is a STREAMS-based module that has open, close, and put procedures.

When the named stream is opened, the open routine of connld(7M) is called. The open fails if:

- The pipe ends cannot be created.
- A file pointer and file descriptor cannot be allocated.
- The stream head cannot stream the two pipe ends.

The open is not complete and will block until the server process has received the file descriptor using the I_RECVFD ioctl. The setting of the O_NDELAY or O_NONBLOCK flag has no impact on the open routine.

connld(7M) does not process messages. All messages are passed to the next object in the stream. The read, write, and put routines call putnext(9F) to send the message up or down the stream.

The connld(7M) module can be pushed onto the named end of the pipe. If the named end of the pipe is then opened by a client, a new pipe is created. One file descriptor for the new pipe

is passed back to a client (named stream) as the file descriptor from open(2) and the other file descriptor is passed to the server using I_RECUFD ioctl(2). The server and the client may then communicate through a new pipe.

Figure 6-3 shows a server process that has created a pipe and pushed the connld module on the other end. The server then invokes the fattach(3C) routine to name the other end /usr/toserv.

FIGURE   6-3   Server Sets Up a Pipe

FIGURE   6-4   Processes X and Y Open /usr/toserv
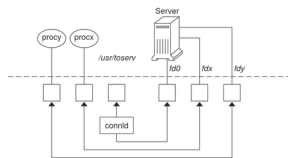
When process X (procx) opens /usr/toserv, it gains a unique connection to the server process that was at one end of the original STREAMS-based pipe. When process Y (procy) does the same, it also gains a unique connection to the server. As shown in Figure 6-4, the server process has access to three separate pipes through three file descriptors.

PART II

Kernel Interface

, STREAMS Framework –Kernel Level	Describes the STREAMS components from the kernel developer's perspective.
, Messages - Kernel Level	Describes the structure and use of each STREAMS message type.
, STREAMS Drivers	Discusses several topics specific to STREAMS device drivers.
, Modules	Provides specific examples of how modules work.
, Configuration	Describes how to configure STREAMS drivers and modules into the Oracle Solaris operating environment.
, MultiThreaded STREAMS	Describes how to multithread a STREAMS driver or module.
, Multiplexing	Describes how STREAMS multiplexing configurations are created and also discusses multiplexing drivers.

7
♦ ♦ ♦ **C H A P T E R   7**

STREAMS Framework – Kernel Level

Because the STREAMS subsystem of UNIX® provides a framework on which communications services can be built, it is often called the STREAMS *framework*. This framework consists of the stream head and a series of utilities (`put`, `putnext`), kernel structures (`mblk`, `dblk`), and linkages (queues) that facilitate the interconnections between modules, drivers, and basic system calls. This chapter describes the STREAMS components from the kernel developer's perspective.

- "Overview of Streams in Kernel Space" on page 83
- "Stream Head" on page 84
- "Kernel–Level Messages" on page 84
- "Message Queues" on page 93
- "Entry Points" on page 96
- "Flow Control in Service Procedures" on page 110

Overview of Streams in Kernel Space

Chapter 1, "Overview of STREAMS" describes a stream as a full-duplex processing and data transfer path between a STREAMS driver in kernel space and a process in user space. In the kernel, a stream consists of a stream head, a driver, and zero or more modules between the driver and the stream head.

The stream head is the end of the stream nearest the user process. All system calls made by user-level applications on a stream are processed by the stream head.

Messages are the containers in which data and control information is passed between the stream head, modules, and drivers. The stream head is responsible for translating the appropriate messages between the user application and the kernel. Messages are simply pointers to structures (`mblk`, `dblk`) that describe the data contained in them. Messages are categorized by type according to the purpose and priority of the message.

Queues are the basic elements by which the stream head, modules, and drivers are connected. Queues identify the `open`, `close`, `put`, and `service` entry points. Additionally, queues specify parameters and private data for use by modules and drivers, and are the repository for messages destined for deferred processing.

In the rest of this chapter, the word "modules" refers to modules, drivers, or multiplexers, except where noted.

Stream Head

The stream head interacts between applications in the user space and the rest of the stream in kernel space. The stream head is responsible for configuring the plumbing of the stream through `open`, `close`, `push`, `pop`, `link`, and `unlink` operations. It also translates user data into messages to be passed down the stream, and translates messages that arrive at the stream head into user data. Any characteristics of the stream that can be modified by the user application or the underlying stream are controlled by the stream head, which also informs users of data arrival and events such as error conditions.

If an application makes a system call with a STREAMS file descriptor, the stream head routines are invoked, resulting in data copying, message generation, or control operations. Only the stream head can copy data between the user space and kernel space. All other parts of the stream pass data by way of messages and are thus isolated from direct interaction with users of the stream.

Kernel–Level Messages

Chapter 3, "STREAMS Application-Level Mechanisms" discusses messages from the application perspective. The following sections discuss message types, message structure and linkage; how messages are sent and received; and message queues and priority from the kernel perspective.

Message Types

Several STREAMS messages differ in their purpose and queueing priority. The message types are briefly described and classified, according to their queueing priority, in Table 7-1 and Table 7-2. A detailed discussion of message types is in Chapter 8, "STREAMS Kernel-Level Mechanisms".

Some message types are defined as high-priority types. Ordinary or normal messages can have a normal priority of 0, or a priority (also called a *band*) from 1 to 255.

TABLE 7-1   Ordinary Messages, Description of Communication Flow

Ordinary Messages	Direction	
M_BREAK	Request to a stream driver to send a "break"	Upstream

Ordinary Messages	Direction	
M_CTL	Control or status request used for intermodule communication	Bidirectional
M_DATA	User data message for I/O system calls	Bidirectional
M_DELAY	Request for a real-time delay on output	Downstream
M_IOCTL	Control/status request generated by a stream head	Downstream
M_PASSFP	File pointer-passing message	Bidirectional
M_PROTO	Protocol control information	Bidirectional
M_SETOPTS	Sets options at the stream head; sends upstream	Upstream
M_SIG	Signal sent from a module or driver	Upstream

TABLE 7-2   High-Priority Messages, Description of Communication Flow

High-Priority Messages	Direction	
M_COPYIN	Copies in data for transparent ioctls	Upstream
M_COPYOUT	Copies out data for transparent ioctls	Upstream
M_ERROR	Reports downstream error condition	Upstream
M_FLUSH	Flushes module queue	Bidirectional
M_HANGUP	Sets a stream head hangup condition	Upstream
M_UNHANGUP	Reconnects line, sends upstream when hangup reverses	Upstream
M_IOCACK	Positive ioctl acknowledgement	Upstream
M_IOCDATA	Data for transparent ioctls, sent downstream	Downstream
M_IOCNAK	Negative ioctl acknowledgement	Upstream
M_PCPROTO	Protocol control information	Bidirectional
M_PCSIG	Sends signal from a module or driver	Upstream
M_READ	Read notification; sends downstream	Downstream
M_START	Restarts stopped device output	Downstream
M_STARTI	Restarts stopped device input	Downstream
M_STOP	Suspends output	Downstream
M_STOPI	Suspends input	Downstream

Message Structure

A STREAMS message in its simplest form contains three elements—a message block, a data block, and a data buffer. The data buffer is the location in memory where the actual data of the message is stored. The data block (datab(9S) describes the data buffer—where it starts, where it ends, the message types, and how many message blocks reference it. The message block (msgb(9S)) describes the data block and how the data is used.

The data block has a typedef of dblk_t and has the following public elements:

```
struct datab {
 unsigned char      *db_base;         /* first byte of buffer */
 unsigned char      *db_lim;          /* last byte+1 of buffer */
 unsigned char       db_ref;          /* msg count ptg to this blk */
 unsigned char       db_type;         /* msg type */
};

typedef struct datab dblk_t;
```

The datab structure specifies the data buffers' fixed limits (db_base and db_lim), a reference count field (db_ref), and the message type field (db_type). db_base points to the address where the data buffer starts, db_lim points one byte beyond where the data buffer ends, and db_ref maintains a count of the number of message blocks sharing the data buffer.

⚠️ **Caution -** db_base, db_lim, and db_ref should not be modified directly. db_type is modified under carefully monitored conditions, such as changing the message type to reuse the message block.

In a simple message, the message block references the data block, identifying for each message the address where the message data begins and ends. Each simple message block refers to the data block to identify these addresses, which must be within the confines of the buffer such that $db\_base \geq b\_rptr \geq\geq b\_wptr \geq db\_lim$. For ordinary messages, a priority band can be indicated, and this band is used if the message is queued.

Figure 7-1 shows the linkages between msgb, datab, and the data buffer in a simple message.

FIGURE 7-1   Simple Message Referencing the Data Block
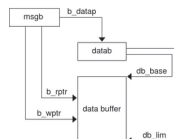
The message block (see msgb(9S)) has a typedef of mblk_t and has the following public elements:

```
struct msgb {
 struct msgb           *b_next;     /*next msg in queue*/
 struct msgb           *b_prev;     /*previous msg in queue*/
 struct msgb           *b_cont;     /*next msg block of message*/
 unsigned char         *b_rptr;     /*1st unread byte in bufr*/
 unsigned char         *b_wptr;     /*1st unwritten byte in bufr*/
 struct datab          *b_datap;    /*data block*/
 unsigned char          b_band;     /*message priority*/
 unsigned short         b_flag;     /*message flags*/
};
```

The STREAMS framework uses the b_next and b_prev fields to link messages into queues. b_rptr and b_wptr specify the current read and write pointers respectively, in the data buffer pointed to by b_datap. The fields b_rptr and b_wptr are maintained by drivers and modules.

The field b_band specifies a priority band where the message is placed when it is queued using the STREAMS utility routines. This field has no meaning for high-priority messages and is set to zero for these messages. When a message is allocated using allocb(9F), the b_band field is initially set to zero. Modules and drivers can set this field to a value from 0 to 255 depending on the number of priority bands needed. Lower numbers represent lower priority. The kernel incurs overhead in maintaining bands if nonzero numbers are used.

> **Caution -** Message block data elements must not modify b_next, b_prev, or b_datap. The first two fields are modified by utility routines such as putq(9F) and getq(9F). Message block data elements can modify b_cont, b_rptr, b_wptr, b_band (for ordinary messages types), and b_flag.

The SunOS environment places b_band in the msgb structure. Some other STREAMS implementations place b_band in the datab structure. The SunOS implementation is more flexible because each message is independent. For shared data blocks, the b_band can differ in the SunOS implementation, but not in other implementations.

Message Linkage

A complex message can consist of several linked message blocks. If buffer size is limited or if processing expands the message, multiple message blocks are formed in the message, as shown in Figure 7-2 . When a message is composed of multiple message blocks, the type associated with the first message block determines the overall message type, regardless of the types of the attached message blocks.

FIGURE   7-2   Linked Message Blocks
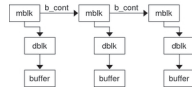
Queued Messages

A `put` procedure processes single messages immediately and can pass the message to the next module's `put` procedure using `put` or `putnext`. Alternatively, the message is linked on the message queue for later processing, to be processed by a module's service procedure (`putq`(9F)). Note that only the first module of a set of linked modules is linked to the next message in the queue.

Think of linked message blocks as a concatenation of messages. Queued messages are a linked list of individual messages that can also be linked message blocks.

FIGURE   7-3   Queued Messages
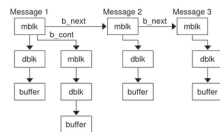
In Figure 7-3 messages are queued: Message 1 being the first message on the queue, followed by Message 2 and Message 3. Notice that Message 1 is a linked message consisting of more than one `mblk`.

⚠️ **Caution -** Modules or drivers must not modify `b_next` and `b_prev`. These fields are modified by utility routines such as `putq`(9F) and `getq`(9F).

Shared Data

In Figure 7-4 , two message blocks are shown pointing to one data block. `db_ref` indicates that there are two references (`mblks`) to the data block. `db_base` and `db_lim` point to an address

range in the data buffer. The `b_rptr` and `b_wptr` of both message blocks must fall within the assigned range specified by the data block.

FIGURE 7-4 Shared Data Block
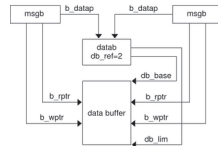
Data blocks are shared using utility routines (see dupmsg(9F) or dupb(9F)). STREAMS maintains a count of the message blocks sharing a data block in the `db_ref` field.

These two `mblks` share the same data and datablock. If a module changes the contents of the data or message type, it is visible to the owner of the message block.

When modifying data contained in the `dblk` or data buffer, if the reference count of the message is greater than one, the module should copy the message using copymsg(9F), free the duplicated message, and then change the appropriate data.

Note - *Hardening Information.* It is assumed that a message with a `db_ref > 1` is a "read-only" message and can be read but not modified. If the module wishes to modify the data, it should first copy the message, and free the original:

```
if ( dbp->db_ref > 1 ) {
 dblk_t *newdbp;

 /* Get a copy of the message */
 newdbp = copymsg(dbp);

 /* Free the original */
 freemsg(dbp);

 /* make sure that we are now using the new dbp */
 dbp = newdbp;
}
```

STREAMS provides utility routines and macros (identified in Appendix B, "Kernel Utility Interface Summary") to assist in managing messages and message queues, and in other areas of module and driver development. Always use utility routines to operate on a message queue or to free or allocate messages. If messages are manipulated in the queue without using the STREAMS utilities, the message ordering can become confused and cause inconsistent results.

> ⚠ **Caution -** Not adhering to the DDI/DKI can result in panics and system crashes.

Sending and Receiving Messages

Among the message types, the most commonly used messages are `M_DATA`, `M_PROTO`, and `M_PCPROTO`. These messages can be passed between a process and the topmost module in a stream, with the same message boundary alignment maintained between user and kernel space. This allows a user process to function, to some degree, as a module above the stream and maintain a service interface. `M_PROTO` and `M_PCPROTO` messages carry service interface information among modules, drivers, and user processes.

Modules and drivers do not interact directly with any interfaces except `open`(2) and `close`(2). The stream head translates and passes all messages between user processes and the uppermost STREAMS module. Message transfers between a process and the stream head can occur in different forms. For example, `M_DATA` and `M_PROTO` messages can be transferred in their direct form by `getmsg`(2) and `putmsg`(2). Alternatively, `write`(2) creates one or more `M_DATA` messages from the data buffer supplied in the call. `M_DATA` messages received at the stream head are consumed by `read`(2) and copied into the user buffer.

Any module or driver can send any message in either direction on a stream. However, based on their intended use in STREAMS and their treatment by the stream head, certain messages can be categorized as upstream, downstream, or bidirectional. For example, `M_DATA`, `M_PROTO`, or `M_PCPROTO` messages can be sent in both directions. Other message types such as `M_IOACK` are sent upstream to be processed only by the stream head. Messages to be sent downstream are silently discarded if received by the stream head. Table 7-1 and Table 7-2 indicate the intended direction of message types.

STREAMS enables modules to create messages and pass them to neighboring modules. `read`(2) and `write`(2) are not enough to enable a user process to generate and receive all messages. In the first place, `read`(2) and `write`(2) are byte-stream oriented with no concept of message boundaries. The message boundary of each service primitive must be preserved so that the start and end of each primitive can be located in order to support service interfaces. Furthermore, `read`(2) and `write`(2) offer only one buffer to the user for transmitting and receiving STREAMS messages. If control information and data is placed in a single buffer, the user has to parse the contents of the buffer to separate the data from the control information. Furthermore, `read`(2) and `write`(2) offer only one buffer to the user for transmitting and receiving STREAMS messages. If control information and data is placed in a single buffer, the user has to parse the contents of the buffer to separate the data from the control information.

`getmsg`(2) and `putmsg`(2) enable a user process and the stream to pass data and control information between one another while maintaining distinct message boundaries.

Data Alignment

Note - *Hardening Information*. There is no guarantee in STREAMS that a `b_rptr` or `b_wptr` will fall on a proper bit alignment. Most modules that pass data structures with pointers try to retain the desired bit alignment. If the module is in a stream where this is reasonably guaranteed, it does not need to check data alignment. However, for the purpose of hardening, modules that are concerned about data alignment should verify that pointers are properly aligned, or copy data in `mblks` to local structures that are properly aligned (see bcopy(3C)).

Note - *Hardening Information*. Ensure that the changing of pointers is uniform (`b_rptr` <= `b_rptr`). Keep pointers inside `db_base` and `db_lim`. It is easier to recover from an error if `b_rptr` and `b_wptr` are inside `db_base` and `db_lim`.

When a module changes the `b_rptr` and/or the `b_wptr`, it should verify the following relationship:

```
db_base <= b_rptr <= b_wptr <= db_lim
```

and

```
db_base < db_lim
```

Message Queues and Message Priority

Message queues grow when the STREAMS scheduler is delayed from calling a service procedure by system activity, or when the procedure is blocked by flow control. When called by the scheduler, a module's service procedure processes queued messages in a FIFO manner (getq(9F)). However, some messages associated with certain conditions, such as `M_ERROR`, must reach their stream destination as rapidly as possible. This is accomplished by associating priorities with the messages. These priorities imply a certain ordering of messages in the queue, as shown in Figure 7-5.

Each message has a priority band associated with it. Ordinary messages have a priority band of zero. The priority band of high-priority messages is ignored since they are high priority and thus not affected by flow control. putq(9F) places high-priority messages at the head of the message queue, followed by priority band messages (expedited data) and ordinary messages.

FIGURE   7-5   Message Ordering in a Queue

When a message is queued, it is placed after the messages of the same priority already in the queue (in other words, FIFO within their order of queueing). This affects the flow-control parameters associated with the band of the same priority. Message priorities range from 0 (normal) to 255 (highest). This provides up to 256 bands of message flow within a stream. An example of how to implement expedited data would be with one extra band of data flow (priority band 1), is shown in the following figure. Queues are explained in detail in the next section.

FIGURE   7-6   Message Ordering with One Priority Band

High-priority messages are not subject to flow control. When they are queued by `putq`(9F), the associated queue is always scheduled, even if the queue has been disabled (`noenable`(9F)).

When the service procedure is called by the stream's scheduler, the procedure uses `getq`(9F) to retrieve the first message on queue, which is a high-priority message. Service procedures must be implemented to act on high-priority messages immediately. The mechanisms just mentioned—priority message queueing, absence of flow control, and immediate processing by a procedure—result in rapid transport of high-priority messages between the originating and destination components in the stream.

In general, high-priority messages should be processed immediately by the module's `put` procedure and not placed on the `service` queue.

⚠️ **Caution -** A `service` procedure must never queue a high-priority message on its own queue because an infinite loop results. The enqueuing triggers the queue to be immediately scheduled again.

Message Queues

The queue is the fundamental component of a stream. It is the interface between a STREAMS module and the rest of the stream, and is the repository for deferred message processing. For each instance of an open driver or pushed module or stream head, a pair of queues is allocated, one for the read side of the stream and one for the write side.

The RD(9F), WR(9F), and OTHERQ(9F) routines allow reference of one queue from the other. Given a queue, RD(9F) returns a pointer to the read queue, WR(9F) returns a pointer to the write queue and OTHERQ(9F) returns a pointer to the opposite queue of the pair (see queue(9S)).

By convention, queue pairs are depicted graphically as side- by- side blocks, with the write queue on the left and the read queue on the right (see following figure).

FIGURE 7-7   Queue Pair Allocation

queue Structure

As previously discussed, messages are ordered in message queues. Message queues, message priority, service procedures, and basic flow control all combine in STREAMS. A service procedure processes the messages in its queue. If there is no service procedure for a queue, putq(9F) does not schedule the queue to be run. The module developer must ensure that the messages in the queue are processed. Message priority and flow control are associated with message queues.

The queue structure is defined in stream.h as a typedef queue_t, and has the following public elements:

```
struct  qinit   *q_qinfo; /* procs and limits for queue */
struct  msgb    *q_first; /* first data block */
struct  msgb    *q_last; /* last data block */
struct  queue   *q_next; /* Q of next stream */
struct  queue   *q_link; /* to next Q for scheduling */
void            *q_ptr;  /* to private data structure */
size_t          q_count; /* number of bytes on Q */
uint            q_flag;  /* queue state */
ssize_t         q_minpsz; /* min packet size accepted by this module */
ssize_t         q_maxpsz; /* max packet size accepted by this module */
size_t          q_hiwat; /* queue high—water mark */
```

```
size_t          q_lowat; /* queue low-water mark */
```

`q_first` points to the first message on the queue, and `q_last` points to the last message on the queue. `q_count` is used in flow control and contains the total number of bytes contained in normal and high-priority messages in band 0 of this queue. Each band is flow controlled individually and has its own count. For more details, see "qband Structure" on page 106. `qsize(9F)` can be used to determine the total number of messages on the queue. `q_flag` indicates the state of the queue. See Table 7-3 for the definitions of these flags.

`q_minpsz` contains the minimum packet size accepted by the queue, and `q_maxpsz` contains the maximum packet size accepted by the queue. These are suggested limits, and some implementations of STREAMS may not enforce them. The SunOS stream head enforces these values but they are voluntary at the module level. You should design modules to handle messages of any size.

`q_hiwat` indicates the limiting maximum number of bytes that can be put on a queue before flow control occurs. `q_lowat` indicates the lower limit where STREAMS flow control is released.

`q_ptr` is the element of the `queue` structure where modules can put values or pointers to data structures that are private to the module. This data can include any information required by the module for processing messages passed through the module, such as state information, module IDs, routing tables, and so on. Effectively, this element can be used any way the module or driver writer chooses. `q_ptr` can be accessed or changed directly by the driver, and is typically initialized in the `open(9E)` routine.

When a queue pair is allocated, `streamtab` initializes `q_qinfo`, and `module_info` initializes `q_minpsz`, `q_maxpsz`, `q_hiwat`, and `q_lowat`. Copying values from the `module_info` structure enables them to be changed in the queue without modifying the `streamtab` and `module_info` values.

The following table lists the `queue(9S)` flags.

TABLE 7-3   Queue Flags

Flag	Description
QENAB	The queue is enabled to run the service procedure
QWANTR	Someone wants to read queue
QWANTW	Someone wants to write to queue
QFULL	The queue is full
QREADR	Set for all read queues
QUSE	This queue in use (allocation)

Flag	Description
QNOENB	Do not enable the queue when data is placed on it

Using Queue Information

The `q_first`, `q_last`, `q_count`, and `q_flags` components must not be modified by the module, and should be accessed using `strqget`(9F). The values of `q_minpsz`, `q_maxpsz`, `q_hiwat`, and `q_lowat` are accessed through `strqget`(9F), and are modified by `strqset`(9F). `q_ptr` can be accessed and modified by the module and contains data private to the module.

All other accesses to fields in the `queue`(9S) structure should be made through STREAMS utility routines (see Appendix B, "STREAMS Utilities"). Modules and drivers should not change any fields not explicitly listed previously.

`strqget`(9F) enables modules and drivers to get information about a queue or particular band of the queue. This insulates the STREAMS data structures from the modules and drivers. The prototype for the `strqget`(9F) routine is:

```
int
strqget(queue_t *q, qfields_t what, unsigned char pri, void *valp)
```

`q` specifies from which queue the information is to be retrieved; `what` defines the `queue_t` field value to obtain (see the following structure fields). `pri` identifies a specific priority band. The value of the field is returned in `valp`. The fields that can be obtained are defined in `<sys/stream.h>` and shown here as:

```
QHIWAT              /* high-water mark */
QLOWAT              /* low-water mark */
QMAXPSZ             /* largest packet accepted */
QMINPSZ             /* smallest packet accepted */
QCOUNT              /* approx. size (in bytes) of data */
QFIRST              /* first message */
QLAST               /* last message */
QFLAG               /* status */
```

`strqset`(9F) enables modules and drivers to change information about a queue or a band of the queue. This also insulates the STREAMS data structures from the modules and drivers. Its prototype is:

```
int
strqset(queue_t *q. qfields_t what, unsigned char pri, intptr_t val)
```

The *q*, *what*, and *pri* fields are the same as in `strqget`(9F), but the information to be updated is provided in *val* instead of through a pointer. If the field is read-only, `EPERM` is returned and the field is left unchanged. The following fields are read-only: `QCOUNT`, `QFIRST`, `QLAST`, and `QFLAG`.

> **Note -** *Hardening Information*. Access queue structure information, through `strqget` and `strqset` only. Do not access the `queue` structure directly.

Entry Points

The `q_qinfo` component points to a `qinit` structure. This structure defines the module's entry point procedures for each queue, which include the following:

```
int      (*qi_putp)();      /* put procedure */
int      (*qi_srvp)();      /* service procedure */
int      (*qi_qopen)();     /* open procedure */
int      (*qi_qclose)();    /* close procedure */
struct module_info *qi_minfo;  /* module information structure */
```

There is generally a unique `q_init` structure associated with the read queue and the write queue. `qi_putp` identifies the `put` procedure for the module. `qi_srvp` identifies the optional `service` procedure for the module.

The `open` and `close` entry points are required for the read-side queue. The `put` procedure is generally required on both queues and the `service` procedure is optional.

> **Note -** *Hardening Information*. If the `put` procedure is not defined and a subsequent `put` is done to the module, a panic occurs. As a precaution, `putnext` should be declared as the module's `put` procedure.
>
> If a module only requires a `service` procedure, `putq`(9F) can be used as the module's `put` procedure. If the `service` procedure is not defined, the module's `put` procedure must not queue data (`putq`).

The `qi_qopen` member of the read-side `qinit` structure identifies the `open`(9E) entry point of the module. The `qi_qclose` member of the read-side `qinit` structure identifies the `close`(9E) entry point of the module.

The `qi_minfo` member points to the `module_info`(9S) structure.

```
struct module_info {
  ushort   mi_idnum;          /* module id number */
  char     *mi_idname;        /* module name */
  ssize_t  mi_minpsz;         /* min packet size accepted */
  ssize_t  mi_maxpsz;         /* max packet size accepted */
  size_t   mi_hiwat;          /* hi-water mark */
  size_t   mi_lowat;          /* lo-water mark */
};
```

`mi_idnum` is the module's unique identifier defined by the developer and used in `strlog`(9F). `mi_idname` is an ASCII string containing the name of the module. `mi_minpsz` is the initial minimum packet size of the queue. `mi_maxpsz` is the initial maximum packet size of the queue. `mi_hiwat` is the initial high–water mark of the queue. `mi_lowat` is the initial low–water mark of the queue.

open Routine

The `open`(9E) routine of a device is called once for the initial `open` of the device, then is called again on subsequent reopens of the stream. Module `open` routines are called once for the initial push onto the stream and again on subsequent reopens of the stream.

FIGURE 7-8   Order of a Module's `open` Procedure
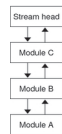
The stream is analogous to a stack. Initially the driver is opened and as modules are pushed onto the stream, their `open` routines are invoked. Once the stream is built, this order reverses if a reopen of the stream occurs. For example, while building the stream shown in Figure 7-8, device A's `open` routine is called, followed by B's and C's when they are pushed onto the stream. If the stream is reopened, Module C's `open` routine is called first, followed by B's, and finally by A's.

Usually the module or driver does not check this, but the issue is raised so that dependencies on the order of `open` routines are not introduced by the programmer. Note that although an `open` can happen more than once, `close` is only called once. See the next section on the close routine for more details. If a file is duplicated (`dup`(2)) the stream is not reopened.

The prototype for the `open` entry point is:

```
int prefix_open(queue_t *q, dev_t *devp, int oflag, int sflag,
     cred_t *cred_p)
```

q   Pointer to the read queue of this module.

devp   Pointer to a device number that is always associated with the device at the end of the stream. Modules cannot modify this value, but drivers can, as described in Chapter 9, "STREAMS Drivers".

oflag	For devices, *oflag* can contain the following bit mask values: `FEXCL`, `FNDELAY`, `FREAD`, and `FWRITE`. See Chapter 9, "STREAMS Drivers" for more information on drivers.
sflag	When the `open` is associated with a driver, *sflag* is set to 0 or `CLONEOPEN`, see Chapter 9, "STREAMS Drivers" , "Cloning STREAMS Drivers" on page 180 for more details. If the open is associated with a module, *sflag* contains the value `MODOPEN`.
cred_p	Pointer to the user credentials structure.

The `open` routines to devices are serialized . If more than one process attempts to open the device, only one proceeds and the others wait until the first finishes. Interrupts are not blocked during an `open`. The driver's interrupt routine must continue to handle interrupts when multiple processes are opening the same device. See Chapter 9, "STREAMS Drivers" for more information.

The `open` routines for both drivers and modules have user context. For example, they can do blocking operations, but the blocking routine should return in the event of a signal. In other words, `q_wait_sig` is allowed, but `q_wait` is not.

If the module or driver is to allocate a controlling terminal, it should send an `M_SETOPTS` message with `SO_ISTTY` set to the stream head.

The `open` routine usually initializes the `q_ptr` member of the queue. `q_ptr` is generally initialized to some private data structure that contains various state information private to the module or driver. The module's `close` routine is responsible for freeing resources allocated by the module including `q_ptr`. The following example shows a simple `open` routine.

EXAMPLE 7-1   A Simple `open` Routine

```
/* example of a module open */
int xx_open(queue_t *q, dev_t *devp, int oflag, int sflag, cred_t *crp)
{
 struct xxstr *xx_ptr;

 xx_ptr = kmemzalloc(sizeof(struct xxstr), KM_SLEEP);
 xx_ptr->xx_foo = 1;
 q->q_ptr = WR(q)->q_ptr = xx_ptr;
 qprocson(q);
 return (0);
}
```

In a multithreaded environment, data can flow up the stream during the `open`. A module receiving this data before its `open` routine finishes initialization can panic. To eliminate this problem, modules and drivers are not linked into the stream until `qprocson`(9F) is called (messages flow around the module). The following figure illustrates this process.

See Chapter 12, "Multithreaded STREAMS" for more information on the multithreaded environment and the use of perimeters.

FIGURE   7-9   Messages Flowing Around the Module Before `qprocson`
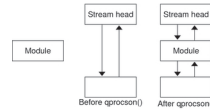
The module or driver instance is guaranteed to be single-threaded before `qprocson`(9F) is called, except for interrupts or callbacks that must be handled separately.

Note - *Hardening Information.* `qprocson` must be called before calling `qbufcall`(9F), `qtimeout`(9F), `qwait`(9F), or `qwait_sig`(9F).

Before a module calls `qprocson`, it must be ready to accept data via the module's `put` procedure so all data structures must be fully initialized (see "put Procedure" on page 101). The most common method for calling `qprocson` is to call this function just before returning from a successful `open` (see Example 7-1).

Note - For a multithreaded environment, verify you are using the correct perimeter before accessing data. See Chapter 12, "Multithreaded STREAMS" for more information on the multithreaded environment and the use of perimeters.

`close` Routine

The `close` routine of devices is called only during the last `close` of the device. Module `close` routines are called during the last `close` or if the module is popped.

The prototype for the close entry point is:

```
int prefix_close (queue *q, int flag, cred_t * cred_p)
```

q   is a pointer to the read queue of the module.

flag   is analogous to the *oflag* parameter of the open entry point. If `FNBLOCK` or `FNDELAY` is set, then the module should attempt to avoid blocking during the close.

cred_p   is a pointer to the user credential structure.

Like open, the close entry point has user context and can block. The blocking routines should return in the event of a signal. Device drivers must take into consideration that interrupts are not blocked during close.

close might be called in a context where a thread cannot receive signals, such as calling close during exit closure of open file descriptors. The system does not reawaken the thread when a user-level process attempts to send a signal, including SIGKILL, to the process.

Use ddi_can_receive_sig to determine whether a thread can receive user-level signals. ddi_can_receive_sig returns B_TRUE if the current thread can receive user-level signals, and B_FALSE if the thread cannot. In this case, qwait_sig behaves exactly like qwait. Use qtimeout or other facilities to prevent close from blocking indefinitely when a thread cannot receive signals.

In particular, asynchronous serial drivers should use caution when draining output data after calling close. Under most conditions, the driver must attempt to wait as long as possible to drain all output data and to discard the data only when a signal is received. However, if ddi_can_receive_sig returns B_FALSE and output flow control is asserted indefinitely by the peer, the driver must abort the drain operation after a reasonable time period has elapsed. Otherwise, the device could remain unusable until the next system boot.

The close routine must cancel all pending and qbufcall callbacks, and process any remaining messages on its service queue. In Example 7-2, the open and close procedures are only used on the read side of the queue and can be set to NULL in the write-side qinit structure initialization.

EXAMPLE 7-2   Example of a Module Close

```
/* example of a module close */
static int
xx_close(queue_t *, *rq, int flag, cred_t *credp)
{
  struct xxstr   *xxp;

   /*
    * Disable xxput() and xxsrv() procedures on this queue.
    */
  qprocsoff(rq);
  xxp = (struct xxstr *) rq->q_ptr;

   /*
    * Cancel any pending timeout.
    * This example assumes that the timeout was issued
    * against the write queue.
    */

  if (xxp->xx_timeoutid != 0) {
   (void) quntimeout(WR(rq), xxp->xx_timeoutid);
```

```
   xxp->xx_timeoutid=0;
    }
   /*
    * Cancel any pending bufcalls.
    * This example assumes that the bufcall was issued
    * against the write queue.
    */
  if (xxp->xx_bufcallid !=0) {
   (void) qunbufcall(WR(rq), xxp->xx_bufcallid);
   xxp->xx_bufcallid = 0;
  }
  rq->q_ptr = WR(rq)->q_ptr = NULL;

   /*
    * Free resources allocated during open
    */
  kmem_free (xxp, sizeof (struct xxstr));
  return (0);
}
```

`qprocsoff` does the inverse operation shown in Figure 7-9. This supports the need for cancelling callbacks before a `qprocsoff`.

`qprocsoff` is typically called at the beginning of the `close` routine. The module can no longer receive messages from adjoining modules. The queue, however, still has pointers to it's adjoining modules and can `putnext`. However, as the queue is no longer inserted into the stream, these messages may be out of order from other messages in the stream, so it is best to process these messages before `qprocsoff`.

`qwait` is used because a module needs to get some response from another module or driver in the STREAM (i.e. a DLPI disconnect message sent downstream). `qwait` and `qwait_sig` must also be called before `qprocsoff` because once the queue is removed from the stream, there will be no way for the reply message to reach the queue.

`put` Procedure

The `put` procedure is the mechanism that other modules use to pass messages into this module. This procedure is called via the `put`or `putnext` routines on behalf of other modules. The queue's `put` procedure is invoked by the preceding module to process a message immediately (see `put(9F)` and `putnext(9F)`). Most modules will have a `put` routine. The common exception is on the read-side of drivers because there will not typically be a module downstream to the driver.

Note - *Hardening Information.* `putnext` is used by adjoining modules to ensure that the next module's queue is intact. Use of `put` cannot guarantee that the queue being called is currently valid and inserted into a stream; you must ensure that the queue is valid when using `put`.

A driver's `put` procedure must do one of the following:

- Process and free the message.
- Process and route the message back upstream.
- Queue the message to be processed by the driver's `service` procedure.

All `M_IOCTL` type messages must be acknowledged through `M_IOACK` or rejected through `M_IOCNACK`. `M_IOCTL` messages should not be freed. Drivers must free any unrecognized message.

A module's `put` procedure must do one of the following as shown in Figure 7-10:

- Process and free the message.
- Process the message and pass it to the next module or driver.
- Queue the message to be processed later by the module's `service` procedure.

Unrecognized messages are passed to the next module or driver. The stream operates more efficiently when messages are processed in the `put` procedure. Processing a message with the service procedure imposes some latency on the message.

FIGURE   7-10   Flow of `put` Procedure
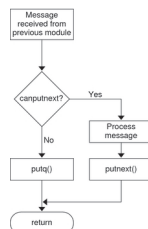
If the next module is flow controlled (see canput(9F) and canputnext(9F)), the `put` procedure can queue the message for processing by the next `service` procedure (see putq(9F)). The `put` routine is always called before the component's corresponding srv(9E) service routine, so always use `put` for immediate message processing.

Note - *Hardening Information.* `canput` and `canputnext` operate similar to `put` and `putnext`; that is the *next* functions verify the integrity of the next queue. Not using the next functions can cause panics as the queue being referenced might have already been closed.

The preferred naming convention for a `put` procedure reflects the direction of the message flow. The read `put` procedure is suffixed by `r` (`rput`), and the write procedure by `w` (`wput`). For example, the read-side `put` procedure for module `xx` is declared as `int xxrput (queue_t *q, mblk_t *mp)`. The write-side `put` procedure is declared as `int xxwput(queue_t *q, mblk_t *mp)`, where *q* points to the corresponding read or write queue and *mp* points to the message to be processed.

Although high-priority messages can be placed on the `service` queue, processing them immediately in the `put` procedure is better. (See the stub code in Example 7-3.) Place ordinary or priority-band messages on the `service` queue (`putq`(9F)) if:

- The stream has been flow controlled; that is, `canput` fails.
- There are already messages on the service queue, that is, `q_first` is not `NULL`.
- Deferred processing is desired.

If other messages already exist on the queue and the `put` procedure does not queue new messages (provided they are not high-priority), messages are reordered. If the next module is flow controlled (see `canput`(9F) and `canputnext`(9F)), the `put` procedure can queue the message for processing by the `service` procedure (see `putq`(9F)).

EXAMPLE 7-3   Example of a Module `put` Procedure

```
/*example of a module put procedure */
int
xxrput(queue_t *,mblk_t, *mp)
{
   /*
    * If the message is a high-priority message or
    * the next module is not flow controlled and we have not
    * already deferred processing, then:
    */

    if (mp->b_datap->db_type >= QPCTL ||
            (canputnext(q) && q->q_first == NULL)) {
       /*
        * Process message
        */


        .
        .
        .
        putnext(q,mp);
    } else {
        /*
         * put message on service queue
         */
         putq(q,mp);
    }
    return (0);
```

```
}
```

Put procedures must never call `putq`, `putbq`, or `qenable` if the module does not have a service procedure.

Note - *Hardening Information.* Once a message is passed using a `putq`, `put`, `putnext`, as well as the perimeter function `qwriter`, it cannot be accessed again because the use of this message has been given to the new routine. If a reference needs to be retained by the module, it should copy it by using `copyb`, `copymsg`, `dupb`, or `dupmsg`.

A module need not process the message immediately, and can queue it for later processing by the `service` procedure (see `putq(9F)`).

The SunOS STREAMS framework is multithreaded. Unsafe (nonmultithreaded) modules are not supported. To make multithreading of modules easier, the SunOS STREAMS framework uses perimeters (see "MT STREAMS Perimeters" on page 230 for more information).

Caution - Mutex locks must not be held across a call to `put(9F)`, `putnext(9F)`, or `qreply(9F)`.

Because of the asynchronous nature of STREAMS, do not assume that a module's `put` procedure has been called just because `put(9F)`, `putnext(9F)`, or `qreply(9F)` has returned.

Queue `service` Procedure

A queue's `service` procedure is invoked to process messages on the queue. It removes successive messages from the queue, processes them, and calls the `put` procedure of the next module in the stream to give the processed message to the next queue.

The `service` procedure is optional. A module or driver can use a `service` procedure for the following reasons:

- Streams flow control is implemented by `service` procedures. If the next component on the stream has been flow controlled, the `put` procedure can queue the message. (See "Flow Control in Service Procedures" on page 110 in Chapter 7, "STREAMS Framework – Kernel Level" for more on flow control.)
- Resource allocation recovery. If a `put` or `service` procedure cannot allocate a resource, such as memory, the message is usually queued to process later.
- A device driver can queue a message and get out of interrupt context.

- To combine multiple messages into larger messages.

The service procedure is invoked by the STREAMS scheduler. A STREAMS service procedure is scheduled to run if:

- The queue is not disabled (noenable(9F)) and the message being queued is either the first message on the queue, or a priority band message.
- The message being queued (putq(9F) or putbq(9F)) is a high-priority message,
- The queue has been back-enabled because flow control has been relieved,
- The queue has been explicitly enabled (qenable(9F)).

A service procedure usually processes all messages on its queue (getq(9F)) or takes appropriate action to ensure it is re-enabled (qenable(9F)) at a later time. Figure 7-11 shows the flow of a service procedure.

Caution - High-priority messages (db_type and MSG_HIPRI) must never be placed back on a service queue (putbq(9F)). Placing these messages in a service queue can cause an infinite loop.

Put procedures must never call putq, putbq, or qenable if the module does not have a service procedure.

FIGURE 7-11   Flow of service Procedure
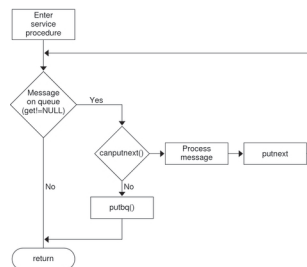
The following example shows the stub code for a module service procedure.

EXAMPLE 7-4   Module service Procedure

```
/*example of a module service procedure */
int
```

```
xxrsrv(queue_t *q)
{
  mblk_t *mp;
   /*
    * While there are still messages on our service queue
    */
  while ((mp = getq(q) != NULL) {
   /*
       * We check for high priority messages, but
       * none is ever seen since the put procedure
       * never queues them.
       * If the next module is not flow controlled, then
       */
      if (mp->b_datap->db_type >= QPCTL || (canputnext (q)) {
     /*
      * process message
      */
     .
     .
     .
     putnext (q, mp);
    } else {
     /*
          * put message back on service queue
          */
     putbq(q,mp);
     break;
    }
  }
  return (0);
}
```

qband Structure

The queue flow information for each band, other than band 0, is contained in a qband(9S) structure. This structure is not visible to other modules. For accessible information see strqget(9F) and strqset(9F). qband is defined as follows:

```
typedef struct qband {
  struct  qband    *qb_next;   /* next band's info */
  size_t  qb_count;            /* number of bytes in band */
  struct  msgb     *qb_first;  /* beginning of band's data */
  struct  msgb     *qb_last;   /* end of band's data */
  size_t  qb_hiwat;            /* high-water mark for band */
  size_t  qb_lowat;            /* low—water mark for band */
  uint    qb_flag;             /* see below */
} qband_t;
```

The structure contains pointers to the linked list of messages in the queue. These pointers, qb_first and qb_last, denote the beginning and end of messages for the particular band. The qb_count field is analogous to the queue's q_count field. However, qb_count only applies to the messages in the queue in the band of data flow represented by the corresponding qband

structure. In contrast, q_count only contains information regarding normal and high-priority messages.

Each band has a separate high and low watermark, qb_hiwat and qb_lowat. These are initially set to the queue's q_hiwat and q_lowat respectively. Modules and drivers can change these values through the strqset(9F) function. The QB_FULL flag for qb_flag denotes that the particular band is full.

The qband(9S) structures are not preallocated per queue. Rather, they are allocated when a message with a priority greater than zero is placed in the queue using putq(9F), putbq(9F), or insq(9F). Since band allocation can fail, these routines return 0 on failure and 1 on success. Once a qband(9S) structure is allocated, it remains associated with the queue until the queue is freed. strqset(9F) and strqget(9F) cause qband(9S) allocation. Sending a message to a band causes all bands up to and including that one to be created.

Using qband Information

The STREAMS utility routines should be used when manipulating the fields in the queue and qband(9S) structures. strqget(9F) and strqset(9F) are used to access band information.

Drivers and modules can change the qb_hiwat and qb_lowat fields of the qband structure. Drivers and modules can only read the qb_count, qb_first, qb_last, and qb_flag fields of the qband structure. Only the fields listed previously can be referenced.

> ⚠ **Caution -** There are fields in the qband structure that are reserved and are not documented. These fields are subject to undocumented, unnotified change at any time.

The following figure shows a queue with two extra bands of flow.

FIGURE 7-12 Data Structure Linkage
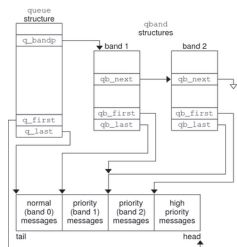
Several routines are provided to aid you in controlling each priority band of data flow. These routines are

- `flushband`(9F)
- `bcanputnext`(9F)
- `strqget`(9F)
- `strqset`(9F)

`flushband`(9F) is discussed in "Flushing Priority Band" on page 149. `bcanputnext`(9F) is discussed in "Flow Control in Service Procedures" on page 110, and the other two routines are described in the following section. Appendix B, "Kernel Utility Interface Summary" also has a description of these routines.

Message Processing Procedures

Typically, `put` procedures are required in pushable modules, but `service` procedures are optional. If the `put` routine queues messages, a corresponding `service` routine must be present to handle the queued messages. If the `put` routine does not queue messages, the `service` routine is not required.

Figure 7-10 shows typical processing flow for a `put` procedure which works as follows:

- A message is received by the `put` procedure associated with the queue, where some processing can be performed on the message.
- The `put` procedure determines if the message can be sent to the next module by the use of `canput`(9F) or `canputnext`(9F).
- If the next module is flow controlled, the `put` procedure queues the message using `putq`(9F).
- `putq`(9F) places the message in the queue based on its priority.
- Then, `putq`(9F) makes the queue ready for execution by the STREAMS scheduler, following all other queues currently scheduled.
- If the next module is not flow controlled, the `put` procedure does any processing needed on the message and sends it to the next module using `putnext`(9F). Note that if the module does not have a `service` procedure it cannot queue the message, and must process and send the message to the next module.

Figure 7-11 shows typical processing flow for a `service` procedure that works as follows:

- When the system goes from kernel mode to user mode, the STREAMS scheduler calls the `service` procedure.

- The `service` procedure gets the first message (`q_first`) from the message queue using the `getq(9F)` utility.

- The `put` procedure determines if the message can be sent to the next module using `canput(9F)` or `canputnext(9F)`.

- If the next module is flow controlled, the `put` procedure requeues the message with `putbq(9F)`, and then returns.

- If the next module is not flow controlled, the `service` procedure processes the message and passes it to the `put` procedure of the next queue with `putnext(9F)`.

- The `service` procedure gets the next message and processes it. This processing continues until the queue is empty or flow control blocks further processing. The `service` procedure returns to the caller.

Caution - A `service` or `put` procedure must never block since it has no user context. It must always return to its caller.

If no processing is required in the `put` procedure, the procedure does not have to be explicitly declared. Rather, `putq(9F)` can be placed in the `qinit(9S)` structure declaration for the appropriate queue side to queue the message for the `service` procedure. For example:

```
static struct qinit winit = { putq, modwsrv, ...... };
```

More typically, `put` procedures process high-priority messages to avoid queueing them.

Device drivers associated with hardware are examples of STREAMS devices that might not have a `put` procedure. Since there are no queues below the hardware level, another module does not call the module's `put` procedure. Data comes into the stream from an interrupt routine, and is either processed or queued for the `service` procedure.

A STREAMS filter is an example of a module without a `service` procedure—messages passed to it are either passed or filtered. Flow control is described in "Flow Control in Service Procedures" on page 110.

The key attribute of a `service` procedure in the STREAMS architecture is delayed processing. When a `service` procedure is used in a module, the module developer is implying that there are other, more time-sensitive activities to be performed elsewhere in this stream, in other streams, or in the system in general.

Note - The presence of a `service` procedure is mandatory if the flow control mechanism is to be utilised by the queue. If you do not implement flow control, queues can overflow and hang the system.

Flow Control in Service Procedures

The STREAMS flow control mechanism is voluntary and operates between the two nearest queues in a stream containing service procedures (see Figure 7-13). Messages are held on a queue only if a service procedure is present in the associated queue.

Messages accumulate on a queue when the queue's service procedure processing does not keep pace with the message arrival rate, or when the procedure is blocked from placing its messages on the following STREAMS component by the flow control mechanism. Pushable modules can contain independent upstream and downstream limits. The stream head contains a preset upstream limit (which can be modified by a special message sent from downstream) and a driver can contain a downstream limit. See `M_SETOPTS` for more information.

Flow control operates as follows:

- Each time a STREAMS message-handling routine (for example, `putq(9F)`) adds or removes a message from a message queue, the limits are checked. STREAMS calculates the total size of all message blocks (`bp->b_wptr - bp->b_rptr`) on the message queue.

Note - `bp` is a pointer to the buffer header structure allocated by `bp_mapin`, `b_wptr` is the first unwritten byte in the buffer, and `b_rptr` is the first unread byte in the buffer. See `msgb(9S)` STREAMS message block structure.

- The total is compared to the queue high and low watermark values. If the total exceeds the high watermark value, an internal full indicator is set for the queue. The operation of the service procedure in this queue is not affected if the indicator is set, and the service procedure continues to be scheduled.
- The next part of flow control processing occurs in the nearest preceding queue that contains a service procedure. In the following figure, if D is full and C has no service procedure, then B is the nearest preceding queue.

FIGURE 7-13 Flow Control Mechanism

- The service procedure in B uses `canputnext(9F)` to check if a queue ahead is marked full. If messages cannot be sent, the scheduler blocks the service procedure in B from further execution. B remains blocked until the low watermark of the full queue, D, is reached.

- While B is blocked, any messages except high-priority messages arriving at B accumulate on its message queue. High-priority messages are not subject to flow control. Eventually, B can reach a full state and the full condition propagates back to the preceding module in the stream.

- When the service procedure processing on D causes the message block total to fall below the low watermark, the full indicator is turned off. STREAMS then schedules the nearest preceding blocked queue (B in this case). This automatic scheduling is called back-enabling a queue.

Modules and drivers need to observe the message priority. High-priority messages, determined by the type of the first block in the message,

```
mp->b_datap->db_type >= QPCTL
```

are not subject to flow control. They should be processed immediately and forwarded, as appropriate.

For ordinary messages, flow control must be tested before any processing is performed. canputnext(9F) determines if the forward path from the queue is blocked by flow control.

This is the general flow control processing of ordinary messages:

- Retrieve the message at the head of the queue with getq(9F).
- Determine if the message type is high priority and not to be processed here.
- If so, pass the message to the put procedure of the following queue with putnext(9F).
- Use canputnext(9F) to determine if messages can be sent onward.
- If messages cannot be forwarded, put the message back in the queue with putbq(9F) and return from the procedure.

Caution - High-priority messages must be processed and not placed back on the queue.

- Otherwise, process the message.

The canonical representation of this processing within a service procedure is:

```
while (getq() != NULL)
 if (high priority message || no flow control) {
  process message
  putnext()
 } else {
  putbq()
  return
 }
```

Expedited data has its own flow control with the same processing method as that of ordinary messages. bcanputnext(9F) provides modules and drivers with a test of flow control in a

priority band. It returns 1 if a message of the given priority can be placed in the queue. It returns 0 if the priority band is flow controlled. If the band does not exist in the queue in question, the routine returns 1.

If the band is flow controlled, the higher bands are not affected. However, lower bands are also stopped from sending messages. Without this, lower priority messages can be passed along ahead of the flow-controlled higher priority messages.

The call bcanputnext(q, 0); is equivalent to the call canputnext(q);.

Note - A service procedure must process all messages in its queue unless flow control prevents this.

A service procedure must continue processing messages from its queue until getq(9F) returns NULL. When an ordinary message is queued by putq(9F), the service procedure is scheduled only if the queue was previously empty, and a previous getq(9F) call returns NULL (that is, the QWANTR flag is set). If there are messages in the queue, putq(9F) presumes the service procedure is blocked by flow control and the procedure is automatically rescheduled by STREAMS when the block is removed. If the service procedure cannot complete processing as a result of conditions other than flow control (for example, no buffers), it must ensure a later return (for example, by bufcall(9F)) or discard all messages in the queue. If this is not done, STREAMS never schedules the service procedure to be run unless the queue's put procedure queues a priority message with putq(9F).

Note - High-priority messages are discarded only if there is already a high-priority message on the stream head read queue. That is, there can be only one high-priority message (PC_PROTO) present on the stream head read queue at any time.

---

putbq(9F) replaces a message at the beginning of the appropriate section of the message queue according to its priority. This might not be the same position at which the message was retrieved by the preceding getq(9F). A subsequent getq(9F) might return a different message.

putq(9F) checks only the priority band in the first message. If a high-priority message is passed to putq with a nonzero b_band value, b_band is reset to 0 before placing the message in the queue. If the message is passed to putq(9F) with a b_band value that is greater than the number of qband(9S)structures associated with the queue, putq(9F) tries to allocate a new qband(9S) structure for each band, up to and including the band of the message.

rmvq and insq work similarly. If you try to insert a message out of order in a queue with insq(9F), the message is not inserted and the routine fails.

putq(9F) does not schedule a queue if noenable(9F) was previously called for the queue. noenable(9F) forces putq(9F) to queue the message when called by this queue, but not to schedule the service procedure. noenable(9F) does not prevent the queue from being scheduled by a flow control back-enable. The inverse of noenable(9F) is enableok(9F).

The service procedure is written using the following algorithm:

```
while ((bp = getq(q)) != NULL) {
 if (queclass (bp) == QPCTL) {
  /* Process the message */
  putnext(q, bp);
  } else if (bcanputnext(q, bp->b_band)) {
  /* Process the message */
  putnext(q, bp);
  } else {
  putbq(q, bp);
  return;
  }
 }
```

If the module or driver ignores priority bands, the algorithm is the same as described in the previous paragraphs, except that canputnext(q) is substituted for bcanputnex(q, bp->b_band).

qenable(9F), another flow-control utility, enables a module or driver to cause one of its queues, or another module's queues, to be scheduled. qenable(9F) can also be used to delay message processing. An example of this is a buffer module that gathers messages in its message queue and forwards them as a single, larger message. This module uses noenable(9F) to inhibit its service procedure and queues messages with its put procedure until a certain byte count or "in queue" time has been reached. When either of these conditions is met, the module calls qenable(9F) to cause its service procedure to run.

Another example is a communication line discipline module that implements end-to-end (for example, to a remote system) flow control. Outbound data is held on the write side message queue until the read side receives a transmit window from the remote end of the network.

Note - STREAMS routines are called at different priority levels. Interrupt routines are called at the interrupt priority of the interrupting device. Service routines are called with interrupts enabled (so that service routines for STREAMS drivers can be interrupted by their own interrupt routines).

8

STREAMS Kernel-Level Mechanisms

This chapter describes the STREAMS kernel-level mechanisms:

`ioctl` Processing

STREAMS is a special type of character device driver that is different from the historical character input/output (I/O) mechanism in several ways.

In the classical device driver, all `ioctl`(2) calls are processed by the single device driver, which is responsible for their resolution. The classical device driver has user context, that is, all data can be copied directly to and from user space.

By contrast, the stream head itself can process some `ioctl`(2) calls (defined in `streamio`(7I)). Generally, STREAMS `ioctl`(2) calls operate independently of any particular module or driver on the stream. This means the valid `ioctl`(2) calls that are processed on a stream change over time, as modules are pushed and popped on the stream. The stream modules have no user context and must rely on the stream head to perform `copyin` and `copyout` requests.

There is no user context in a module or driver when the information associated with the `ioctl`(2) call is received. This prevents use of `ddi_copyin`(9F) or `ddi_copyout`(9F) by the module. No user context also prevents the module and driver from associating any kernel data with the currently running process. In any case, by the time the module or driver receives the `ioctl`(2) call, the process generating can have exited.

STREAMS enables user processes to control functions on specific modules and drivers in a stream using `ioctl(2)` calls. In fact, many `streamio(7I)` `ioctl(2)` commands go no further than the stream head. They are fully processed there and no related messages are sent downstream. For an `I_STR` `ioctl(2)` or an unrecognized `ioctl(2)` command, the stream head creates an `M_IOCTL` message, which includes the `ioctl(2)` argument. This is then sent downstream to be processed by the pertinent module or driver. STREAMS ensures that there is only one user-driven `M_IOCTL` operating on a stream at a time. The `M_IOCTL` message is the precursor message type carrying `ioctl(2)` information to modules. Other message types are used to complete the `ioctl` processing in the stream. Each module has its own set of `M_IOCTL` messages it must recognize.

⚠ **Caution -** *Hardening Information*. Modules and drivers should never assume that user data is correct. Users might be able to pass offsets that exceed the buffers supplied, or data that might be in kernel space. Values should always be checked against the range of data that is requested or supplied. Otherwise, panics or data corruption may occur.

Message Allocation and Freeing

The `allocb(9F)` utility routine allocates a message and the space to hold the data for the message. `allocb(9F)` returns a pointer to a message block containing a data buffer of at least the size requested, providing there is enough memory available. The routine returns `NULL` on failure. `allocb(9F)` always returns a message of type `M_DATA`. The type can then be changed if required. `b_rptr` and `b_wptr` are set to `db_base` (see `msgb(9S)` and `datab(9S))`, which is the start of the memory location for the message buffer.

Note - STREAMS often provides buffers that are bit aligned, but there is no guarantee that `db_base` or `db_lim` reside on bit-aligned boundaries. If bit or page alignment is required on module-supplied buffers use `esballoc` For more information about `esballoc` see "Extended STREAMS Buffers" on page 121.

`allocb(9F)` can return a buffer larger than the size requested. If `allocb(9F)` indicates that buffers are not available (`allocb(9F)` fails), the `put` or `service` procedure cannot block to wait for a buffer to become available. Instead, `bufcall(9F)` defers processing in the module or the driver until a buffer becomes available.

If message space allocation is done by the `put` procedure and `allocb(9F)` fails, the message is usually discarded. If the allocation fails in the `service` routine, the message is returned to

the queue. `bufcall`(9F) is called to set a call to the `service` routine when a message buffer becomes available, and the `service` routine returns.

`freeb`(9F) releases the message block descriptor and the corresponding data block, if the reference count (see `datab`(9S)) is equal to 1. If the reference count exceeds 1, the data block is not released.

`freemsg`(9F) releases all message blocks in a message. It uses `freeb`(9F) to free all message blocks and corresponding data blocks.

In Example 8-1, `allocb`(9F) is used by the `bappend` subroutine that appends a character to a message block.

EXAMPLE 8-1   Use of `allocb`

```
/*
 * Append a character to a message block.
 * If (*bpp) is null, it will allocate a new block
 * Returns 0 when the message block is full, 1 otherwise
 */
#define MODBLKSZ  128   /* size of message blocks */

static int bappend(mblk_t **bpp, int ch)
{
  mblk_t *bp;

  if ((bp = *bpp) != NULL) {
    if (bp->b_wptr >= bp->b_datap->db_lim)
     return (0);
  } else {
    if ((*bpp = bp = allocb(MODBLKSZ, BPRI_MED)) == NULL)
     return (1);
  }
  *bp->b_wptr++ = ch;
  return 1;
}
```

`bappend` receives a pointer to a message block and a character as arguments. If a message block is supplied (`*bpp != NULL`), `bappend` checks if there is room for more data in the block. If not, it fails. If there is no message block, a block of at least `MODBLKSZ` is allocated through `allocb`(9F).

If `allocb`(9F) fails, `bappend` returns success and discards the character. If the original message block is not full or the `allocb`(9F) is successful, `bappend` stores the character in the block.

Example 8-2 shows the processing of all the message blocks in any downstream data (type `M_DATA`) messages. `freemsg`(9F) frees messages.

EXAMPLE 8-2 Subroutine `modwput`

```
/* Write side put procedure */
static int modwput(queue_t *q, mblk_t *mp)
{
  switch (mp->b_datap->db_type) {
  default:
    putnext(q, mp);   /* Don't do these, pass along */
    break;

 case M_DATA: {
    mblk_t *bp;
   struct mblk_t *nmp = NULL, *nbp = NULL;

   for (bp = mp; bp != NULL; bp = bp->b_cont) {
     while (bp->b_rptr < bp->b_wptr) {
       if (*bp->b_rptr == '\n')
         if (!bappend(&nbp, '\r'))
          goto newblk;
       if (!bappend(&nbp, *bp->b_rptr))
         goto newblk;

      bp->b_rptr++;
       continue;

    newblk:
      if (nmp == NULL)
        nmp = nbp;
      else { /* link msg blk to tail of nmp */
        linkb(nmp, nbp);
        nbp = NULL;
      }
    }
   }
   if (nmp == NULL)
    nmp = nbp;
   else
    linkb(nmp, nbp);
   freemsg(mp); /* de-allocate message */
   if (nmp)
    putnext(q, nmp);
   break;
   }
  }
}
```

Data messages are scanned and filtered. `modwput` copies the original message into new blocks, modifying as it copies. `nbp` points to the current new message block. `nmp` points to the new message being formed as multiple `M_DATA` message blocks. The outer `for` loop goes through each message block of the original message. The inner `while` loop goes through each byte. `bappend` is used to add characters to the current or new block. If `bappend` fails, the current new block is full. If `nmp` is `NULL`, `nmp` is pointed at the new block. If `nmp` is not `NULL`, the new block is linked to the end of `nmp` by use of `linkb(9F)`.

At the end of the loops, the final new block is linked to nmp. The original message (all message blocks) is returned to the pool by freemsg(9F). If a new message exists, it is sent downstream.

Recovering From No Buffers

bufcall(9F) can be used to recover from an allocb(9F) failure. The call syntax is as follows:

```
bufcall_id_t bufcall(int size, int pri, void(*func)(), long arg);
```

Note - qbufcall(9F) and qunbufcall(9F) must be used with perimeters.

bufcall(9F) calls (*func)(arg) when a buffer of *size* bytes is available. When *func* is called, it has no user context and must return without blocking. Also, there is no guarantee that when *func* is called, a buffer will still be available.

On success, bufcall returns a nonzero identifier that can be used as a parameter to unbufcall(9F) to cancel the request later. On failure, 0 is returned and the requested function is never called.

Caution - Care must be taken to avoid deadlock when holding resources while waiting for bufcall to call (*func*(*arg*). bufcall should be used sparingly.

Read Device Interrupt Handler

Example 8-3 is an example of a read device interrupt handler.

EXAMPLE 8-3 Device Interrupt Handler

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/stream.h>
buffcall_id_t id;      /* hold id val for unbufcall */

dev_rintr(dev)
{
  /* process incoming message ... */
  /* allocate new buffer for device */
  dev_re_load(dev);
}
```

```
/*
 * Reload device with a new receive buffer
 */
dev_re_load(dev)
{
  mblk_t *bp;
  id = 0;       /* begin with no waiting for buffers */
  if ((bp = allocb(DEVBLKSZ, BPRI_MED)) == NULL) {
    cmn_err(CE_WARN,"dev:allocbfailure(size%d)\n",
      DEVBLKSZ);
    /*
     * Allocation failed. Use bufcall to
     * schedule a call to ourselves.
     */
    id = bufcall(DEVBLKSZ,BPRI_MED,dev_re_load,dev);
    return;
  }

  /* pass buffer to device ... */
}
```

See Chapter 12, "Multithreaded STREAMS" for more information on the uses of
unbufcall(9F). These references to unbufcall are protected by MT locks.

Because bufcall(9F) can fail, there is still a chance that the device will hang. A better
strategy if bufcall(9F) fails is to discard the current input message and resubmit that buffer to
the device. Losing input data is preferable to the device hanging.

Write Service Procedure

Example 8-4 is an example of a write service procedure.

EXAMPLE 8-4   Write Service Procedure

```
static int mod_wsrv(queue_t *q)
{
  extern int qenable();
  mblk_t *mp, *bp;
  while (mp = getq(q)) {
   /* check for priority messages and canput ... */

   /* Allocate a header to prepend to the message.
    * If the allocb fails, use bufcall to reschedule.
    */
   if ((bp = allocb(HDRSZ, BPRI_MED)) == NULL) {
    if (!(id=bufcall(HDRSZ,BPRI_MED,qenable, q))) {
      timeout(qenable, (caddr_t)q,
     drv_usectohz());
     /*
      * Put the msg back and exit, we will be
      * re-enabled later
```

```
     */
    putbq(q, mp);
    return;
  }
  /* process message .... */
 }
 }
}
```

`mod_wsrv` prefixes each output message with a header.

In this example, `mod_wsrv` illustrates a potential deadlock case. If `allocb`(9F) fails, `mod_wsrv` tends to recover without loss of data and calls `bufcall`(9F). In this case, the routine passed to `bufcall`(9F) is `qenable`(9F). When a buffer is available, the `service` procedure is automatically re-enabled. Before exiting, the current message is put back in the queue. Example 8-4 deals with `bufcall`(9F) failure by calling `timeout`(9F).

`timeout`(9F) schedules the given function to be run with the given argument in the given number of clock cycles. In this example, if `bufcall`(9F) fails, the system runs `qenable`(9F) after two seconds have passed.

Releasing Callback Requests

When `allocb`(9F) fails and `bufcall`(9F) is called, a callback is pending until a buffer is actually returned. Because this callback is asynchronous, it must be released before all processing is complete. To release this queued event, use `unbufcall`(9F).

Pass the `id` returned by `bufcall`(9F) to `unbufcall`(9F). Then close the driver in the normal way. If this sequence of `unbufcall`(9F) and `xxclose` is not followed, the callback can occur when the driver is already closed. This is one of the most difficult types of problems to find and debug.

Caution - All `bufcall`(9F) and timeouts must be canceled in the close routine.

Extended STREAMS Buffers

Some hardware using the STREAMS mechanism supports memory-mapped I/O (see `mmap`(2)) which allows the sharing of buffers between users, the kernel, and the I/O card. Modules and drivers that need bit-aligned or page-aligned buffers should use extended STREAMS buffers by calling `esballoc` (also see `ddi_umem_alloc`).

If the hardware supports memory-mapped I/O, data received from the hardware is placed in the DARAM (dual-access RAM) section of the I/O card. Since DARAM is memory that is shared between the kernel and the I/O card, coordinated data transfer between the kernel and the I/O card is eliminated. Once in kernel space, the data buffer is manipulated as if it were a kernel resident buffer. Similarly, data sent downstream is placed in the DARAM and forwarded to the network.

In a typical network arrangement, data is received from the network by the I/O card. The controller reads the block of data into the card's internal buffer. It interrupts the host computer to notify that data have arrived. The STREAMS driver gives the controller the kernel address where the data block is to go and the number of bytes to transfer. After the controller has read the data into its buffer and verified the checksum, it copies the data into main memory to the address specified by the DMA (direct memory access) memory address. Once in the kernel space, the data is packaged into message blocks and processed in the usual manner.

When data is transmitted from a user process to the network, it is copied from the user space to the kernel space, packaged as a message block, and sent to the downstream driver. The driver interrupts the I/O card, signaling that data is ready to be transmitted to the network. The controller copies the data from the kernel space to the internal buffer on the I/O card, and from there it is placed on the network.

The STREAMS buffer allocation mechanism enables the allocation of message and data blocks to point directly to a client-supplied (non-STREAMS) buffer. Message and data blocks allocated this way are indistinguishable from the normal data blocks. The client-supplied buffers are processed as if they were normal STREAMS data buffers.

Drivers can attach non-STREAMS data buffers and also free them. This is done as follows:

- Allocation - If the drivers use DARAM without using STREAMS resources and without depending on upstream modules or need to use privately allocated buffers, a data and message block can be allocated without an allocated data buffer. Use `esballoc`(9F). This returns a message block and data block without an associated STREAMS buffer. The buffer used is the one supplied by the caller in the calling sequence.
- Freeing - Each driver using non-STREAMS resources in a STREAMS environment must manage those resources completely, including freeing them. To make this as transparent as possible, a driver-dependent routine is executed if `freeb`(9F) is called to free a message and data block with an attached non-STREAMS buffer.

`freeb`(9F) detects when a buffer is a client supplied, non-STREAMS buffer. If it is, `freeb`(9F) finds the `free_rtn`(9S) structure associated with the buffer. After calling the driver-dependent routine (defined in `free_rtn`(9S)) to free the buffer, `freeb`(9F) frees the message and data block.

The `free` routine should not reference any dynamically allocated data structures that are freed when the driver is closed, as messages can exist in a stream after the driver is closed. For example, when a stream is closed, the driver `close` routine is called and its private data

structure can be deallocated. If the driver sends a message created by `esballoc` upstream, that message can still be on the stream head read queue. When the stream head read queue is flushed, the message is freed and a call is made to the driver's `free` routine after the driver has been closed.

The format of the `free_rtn`(9S) structure is as follows:

```
void (*free_func)();   /*driver dependent free routine*/
char *free_arg;        /* argument for free_rtn */
```

The structure has two fields: a pointer to a function and a location for any argument passed to the function. Instead of defining a specific number of arguments, `free_arg` is defined as a `char *`. This way, drivers can pass pointers to structures if more than one argument is needed.

The method by which `free_func` is called is implementation-specific. Do not assume that `free_func` is called directly from STREAMS utility routines like `freeb`(9F). The `free_func` function must not call another module's `put` procedure nor try to acquire a private module lock that can be held by another thread across a call to a STREAMS utility routine that could free a message block. Otherwise, lock recursion and deadlock could occur.

`esballoc`(9F), provides a common interface for allocating and initializing data blocks. It makes the allocation as transparent to the driver as possible and provides a way to modify the fields of the data block, since modification should only be performed by STREAMS. The driver calls this routine to attach its own data buffer to a newly allocated message and data block. If the routine successfully completes the allocation and assigns the buffer, it returns a pointer to the message block. The driver is responsible for supplying the arguments to `esballoc`(9F), a pointer to its data buffer, the size of the buffer, the priority of the data block, and a pointer to the `free_rtn` structure. All arguments should be non-`NULL`. See Appendix B, "Kernel Utility Interface Summary", for a description of `esballoc`(9F).

`esballoc(9F)` Example

Example 8-5 (which will not compile) shows how extended buffers are managed in the multithreaded environment. The driver maintains a pool of special memory that is allocated by `esballoc`(9F). The allocator `free` routine uses the queue struct assigned to the driver or other queue private data, so the allocator and the close routine need to coordinate to ensure that no outstanding `esballoc`(9F) memory blocks remain after the close. The special memory blocks are of type `ebm_t`, the counter is `ebm`, and the mutex `mp` and the condition variable `cvp` are used to implement the coordination.

EXAMPLE 8-5   `esballoc` Example

```
ebm_t *
```

```
special_new()
{
  mutex_enter(&mp);
  /*
   * allocate some special memory
   */
  esballoc();
  /*
   * increment counter
   */
  ebm++;
  mutex_exit(&mp);
}

void
special_free()
{
  mutex_enter(&mp);
  /*
   * de-allocate some special memory
   */
  freeb();

  /*
   * decrement counter
   */
  ebm--;
  if (ebm == 0)
   cv_broadcast(&cvp);
  mutex_exit(&mp);
}

open_close(q, .....)
 ....
{
  /*
   * do some stuff
   */
  /*
   * Time to decommission the special allocator.  Are there
   * any outstanding allocations from it?
   */
  mutex_enter(&mp);
  while (ebm > 0)
   cv_wait(&cvp, &mp);

  mutex_exit(&mp);
}
```

Caution - The close routine must wait for all esballoc(9F) memory to be freed.

General `ioctl` Processing

Note - Please see the `ioctl` section in the "Writing Device Drivers for Oracle Solaris 11.2 " for information on the 64–bit data structure macros.

When the stream head is called to process an `ioctl`(2) that it does not recognize, it creates an `M_IOCTL` message and sends it down the stream. An `M_IOCTL` message is a single `M_IOCTL` message block followed by zero or more `M_DATA` blocks. The `M_IOCTL` message block has the form of an `iocblk`(9S) structure. This structure contains the following elements.

```
int       ioc_cmd;              /* ioctls command type */
cred_t    *ioc_cr;              /* full credentials */
uint      ioc_id;              /* ioctl id */
uint      ioc_count;           /* byte cnt in data field */
int       ioc_error;           /* error code */
int       ioc_rval;            /* return value */
```

For an `I_STR` `ioctl`(2), `ioc_cmd` contains the command supplied by the user in the `ic_cmd` member of the `strioctl` structure defined in `streamio`(7I). For others, `ioc_cmd` contains the value of the `cmd` argument in the call to `ioctl`(2). The `ioc_cr` field contains the credentials of the user process.

The `ioc_id` field is a unique identifier used by the stream head to identify the `ioctl` and its response messages.

The `ioc_count` field indicates the number of bytes of data associated with this `ioctl` request. If the value is greater than zero, there will be one or more `M_DATA` `mblks` linked to the `M_IOCTL` `mblkb_cont` field. If the value of the `ioc_count` field is zero, there will be no `M_DATA` `mblk` associated with the `M_IOCTL` `mblk`. If the value of `ioc_count` is equal to the special value `TRANSPARENT`, then there is one `M_DATA` `mblk` linked to this `mblk` and its contents will be the value of the argument passed to `ioctl(2)`. This can be a user address or numeric value. (see "Transparent `ioctl` Processing" on page 60).

An `M_IOCTL` message is processed by the first module or driver that recognizes it. If a module does not recognize the command, it should pass it down. If a driver does not recognize the command, it should send a negative acknowledgement or `M_IOCNAK` message upstream. In all circumstances, a module or driver processing an `M_IOCTL` message must acknowledge it.

Modules must always pass unrecognized messages on. Drivers should negatively acknowledge unrecognized `ioctl`(2) messages and free any other unrecognized message.

If a module or driver finds an error in an `M_IOCTL` message for any reason, it must produce a negative acknowledgement message. To do this, set the message type to `M_IOCNAK` and send

the message upstream. No data or return value can be sent. If `ioc_error` is set to 0, the stream head causes the `ioctl(2)` to fail with `EINVAL`. Optionally, the module can set `ioc_error` to an alternate error number.

`ioc_error` can be set to a nonzero value in both `M_IOCACK` and `M_IOCNAK`. This causes the value to be returned as an error number to the process that sent the `ioctl(2)`.

If a module checks what the `ioctl(2)` of other modules below it are doing, the module should not just search for a specific `M_IOCTL` on the write side, but also look for `M_IOCACK` or `M_IOCNAK` on the read side. For example, suppose the module's write side sees `TCSETA` (see `termio(7I)`) and records what is being set. The read-side processing knows that the module is waiting for an answer for the `ioctl(2)`. When the read-side processing sees an `ack` or `nak`, it checks for the same `ioctl(2)` by checking the command (here `TCSETA`) and the `ioc_id`. If these match, the module can use the information previously saved.

If you have the module check, for example, the `TCSETA`/`TCGETA` group of `ioctl(2)` calls as they pass up or down a stream, you must never assume that because `TCSETA` comes down it actually has a data buffer attached to it. The user can form `TCSETA` as an `I_STR` call and accidentally give a `NULL` data buffer pointer. Always check `b_cont` to see if it is `NULL` before using it as an index to the data block that goes with `M_IOCTL` messages.

The `TCGETA` call, if formed as an `I_STR` call with a data buffer pointer set to a value by the user, always has a data buffer attached to `b_cont` from the main message block. Do not assume that the data block is missing and allocate a new buffer, then assign `b_cont` to point to it, because the original buffer will be lost.

STREAMS `ioctl` Issues

Regular device drivers have user context in the `ioctl(9E)` call. However, in a STREAMS driver or module, the only guarantee of user context is in the `open(9E)` and `close(9E)` routines. Some indication of the calling context where data is used is therefore necessary.

Note - The notion of data models as well as new macros for handling data structure access are discussed in "Writing Device Drivers for Oracle Solaris 11.2 ". A STREAMS driver or module writer should use these flags and macros when dealing with structures that change size between data models.

A flag value that represents the data model of the entity invoking the operation has been added to the `ioc_flag` field of the `iocblk(9S)` structure, the `cq_flag` of the `copyreq(9S)` structure, and the `cp_flag` of the `copyresp(9S)` structure.

The data model flag is one of these possibilities:

```
IOC_ILP32
IOC_LP64
```

In addition, IOC_NATIVE is conditionally defined to match the data model of the kernel implementation.

By looking at the data model flag field of the relevant iocblk(9S), copyreq(9S), or copyresp(9S) structures, the STREAMS module can determine the best method of handling the data.

> **Caution -** The layout of the iocblk, copyreq, and copyresp structures is different between the 32-bit and 64-bit kernels. Be cautious of any data structure overloading in the cp_private, cq_private, or cq_filler fields because alignment has changed.

I_STR ioctl Processing

The transparent and nontransparent methods implement ioctl(2) in the STREAMS driver or module itself, rather than in the stream head. I_STR ioctl(2) (also referred to as nontransparent ioctl(2)) is created when a user requests an I_STR ioctl(2) and specifies a pointer to a strioctl structure as the argument. For example, assuming that fd is an open lp STREAMS device and LP_CRLF is a valid option, the user could make a request by issuing the struct in the following example:

EXAMPLE 8-6   Struct for Nontransparent ioctl

```
struct strioctl *str;
short lp_opt = LP_CRLF;

str.ic_cmd = SET_OPTIONS;
str.ic_timout = -1;
str.ic_dp = (char *)&lp_opt;
str.ic_len = sizeof (lp_opt)

ioctl(fd, I_STR, &str);
```

On receipt of the I_STR ioctl(2) request, the stream head creates an M_IOCTL message. ioc_cmd is set to SET_OPTIONS, ioc_count is set to the value contained in ic_len (in this example sizeof (short)). An M_DATA mblk is linked to the M_IOCTL mblk and the data pointed to by ic_dp is copied into it (in this case LP_CRLF).

Example 8-7 illustrates processing associated with an I_STR ioctl(2). lpdoioctl illustrates driver M_IOCTL processing, which also applies to modules. lpdoioctl is called by lp write-side put or service procedure to process M_IOCTL messages. This example is for a driver.

EXAMPLE 8-7   I_STR ioctl(2) Driver

```
static void
lpdoioctl (queue_t *q, mblk_t  *mp)
{
  struct iocblk *iocp;
  struct lp *lp;

  lp = (struct lp *)q->q_ptr;

  /* 1st block contains iocblk structure */
  iocp = (struct iocblk *)mp->b_rptr;

  switch (iocp->ioc_cmd) {
    case SET_OPTIONS:
     /* Count should be exactly one short's worth
      * (for this example) */
     if (iocp->ioc_count != sizeof(short))
      goto iocnak;
     if (mp->b_cont == NULL)
      goto lognak; /* not shown in this example */
     /* Actual data is in 2nd message block */
     iocp->ioc_error = lpsetopt (lp, *(short *)mp->b_cont->b_rptr)

     /* ACK the ioctl */
     mp->b_datap->db_type = M_IOCACK;
     iocp->ioc_count = 0;
     qreply(q, mp);
     break;
    default:
     iocnak:
     /* NAK the ioctl */
     mp->b_datap->db_type = M_IOCNAK;
     qreply(q, mp);
  }
 }
```

This example recognizes only one command, SET_OPTIONS. The ioc_count contains the number of user-supplied data bytes. ioc_count must equal the size of a short.

```
  switch (iocp->ioc_cmd) {
 case SET_OPTIONS:
  /* Count should be exactly one short's worth
   * (for this example) */
  if (iocp->ioc_count != sizeof(short))
   goto iocnak;
  if (mp->b_cont == NULL)
   goto lognak; /* not shown in this example */
```

Once the command has been verified, `lpsetopt` is called to process the request. `lpsetopt` returns 0 if the request is satisfied, otherwise an error number is returned. If `ioc_error` is nonzero, on receipt of the acknowledgement the stream head returns -1 to the application's `ioctl(2)` request and sets `errno` to the value of `ioc_error`.

```
      /* Actual data is in 2nd message block */
  iocp->ioc_error = lpsetopt (lp, *(short *)mp->b_cont->b_rptr)
```

The `ioctl(2)` is acknowledged. This includes changing the `M_IOCTL` message type to `M_IOCACK` and setting the `ioc_count` field to zero to indicate that no data is to be returned to the user. Finally, the message is sent upstream using `qreply(9F)`.

If `ioc_count` was left nonzero, the stream head would copy that many bytes from the second through the *n*th message blocks into the user buffer. You must set `ioc_count` if you want to pass any data back to the user.

```
        /* ACK the ioctl */
  mp->b_datap->db_type = M_IOCACK;
  iocp->ioc_count = 0;
  qreply(q, mp);
  break;
```

In the default case for unrecognized commands or malformed requests, a `nak` is generated. This is done by changing the message type to an `M_IOCNAK` and sending it back upstream.

```
      default:
  iocnak:
  /* NAK the ioctl */
  mp->b_datap->db_type = M_IOCNAK;
  qreply(q, mp);
```

A module does not acknowledge (`nak`) an unrecognized command, but passes the message on. A module does not acknowledge (`nak`) a malformed request.

Transparent `ioctl`

Transparent `ioctl`s are used from within a module to tell the stream head to perform a `copyin` or `copyout` on behalf of the module. The stream head must have knowledge of the data model of the caller in order to process the `copyin` and `copyout` properly. The user should use the `ioctl` macros as described in "Writing Device Drivers for Oracle Solaris 11.2 " when coding a STREAMS module that uses Transparent `ioctl`s.

Transparent `ioctl` Messages

The transparent STREAMS `ioctl`(2) mechanism is needed because user context does not exist in modules and drivers when an `ioctl`(2) is processed. This prevents them from using the kernel `ddi_copyin`/`ddi_copyout` functions.

Transparent `ioctl`(2) enables you to write an application using conventional `ioctl`(2) semantics instead of the `I_STR` `ioctl`(2) and an `strioctl` structure. The difference between transparent and nontransparent `ioctl`(2) processing in a STREAMS driver and module is the way data is transferred from user to kernel space.

The transparent `ioctl`(2) mechanism allows backward compatibility for older programs. This transparency only works for modules and drivers that support transparent `ioctl`(2). Trying to use transparent `ioctl`(2) on a stream that does not support them makes the driver send an error message upstream, causing the `ioctl` to fail.

The following example illustrates the semantic difference between a nontransparent and transparent `ioctl`(2). A module that translates arbitrary character is pushed on the stream The `ioctl`(2) specifies the translation to do (in this case all uppercase vowels are changed to lowercase). A transparent `ioctl`(2) uses `XCASE` instead of `I_STR` to inform the module directly.

Assume that `fd` points to a STREAMS device and that the conversion module has been pushed onto it. The semantics of a nontransparent `I_STR` command to inform the module to change the case of AEIOU are:

```
strioctl.ic_cmd = XCASE;
strioctl.ic_timout = 0;
strioctl.ic_dp = "AEIOU"
strioctl.ic_len = strlen(strioctl.ic_dp);
ioctl(fd,I_STR, &strioctl);
```

When the stream head receives the `I_STR` `ioctl`(2) it creates an `M_IOCTL` message with the `ioc_cmd` set to `XCASE` and the data specified by `ic_dp`. `AEIOU` is copied into the first `mblk` following the `M_IOCTL` `mblk`.

The same `ioctl`(2) specified as a transparent `ioctl`(2) is called as follows:

```
ioctl(fd, XCASE, "AEIOU");
```

The stream head creates an `M_IOCTL` message with the `ioc_cmd` set to `XCASE`, but the data is not copied in. Instead, `ioc_count` is set to `TRANSPARENT` and the address of the user data is placed in the first `mblk` following the `M_IOCTL` `mblk`. The module then requests the stream head to copy in the data (`"AEIOU"`) from user space.

Unlike the nontransparent `ioctl`(2), which can specify a timeout parameter, transparent `ioctl`(2)s block until processing is complete.

> ⚠️ **Caution -** Incorrectly written drivers can cause applications using transparent `ioctl`(2) to block indefinitely.

Even though this process is simpler in the application, transparent `ioctl` adds considerable complexity to modules and drivers, and additional overhead to the time required to process the request.

The form of the `M_IOCTL` message generated by the stream head for a transparent `ioctl`(2) is a single `M_IOCTL` message block followed by one `M_DATA` block. The form of the `iocblk`(9S) structure in the `M_IOCTL` block is the same as described under general `ioctl`(2) processing. However, `ioc_cmd` is set to the value of the command argument in `ioctl`(2) and `ioc_count` is set to the special value of `TRANSPARENT`. The value `TRANSPARENT` distinguishes when an `I_STR` `ioctl`(2) can specify a value of `ioc_cmd` that is equivalent to the `command` argument of a transparent `ioctl`(2). The `b_cont` block of the message contains the value of the `arg` parameter in the call.

> ⚠️ **Caution -** If a module processes a specific `ioc_cmd` and does not validate the `ioc_count` field of the `M_IOCTL` message, the module breaks when transparent `ioctl`(2) is performed with the same command.

> **Note -** Write modules and drivers to support both transparent and `I_STR` `ioctl`(2).

All `M_IOCTL` message types (`M_COPYIN`, `M_COPYOUT`, `M_IOCDATA`,`M_IOCACK` and `M_IOCNACK`) have some similar data structures and sizes. You can reuse these structures instead of reallocating them. Note the similarities in the command type, credentials, and `id`.

The `iocblk`(9S) structure is contained in `M_IOCTL`, `M_IOCACK` and `M_IOCNAK` message types. For the transparent case, `M_IOCTL` has one `M_DATA` message linked to it. This message contains a copy of the argument passed to `ioctl`(2). Transparent processing of `M_IOCACK` and `M_IONAK` does not allow any messages to be linked to them.

The `copyreq`(9S) structure is contained in `M_COPYIN` and `M_COPYOUT` message types. The `M_COPYIN` message type must not have any other message linked to it (that is, `b_cont == NULL`). The `M_COPYOUT` message type must have one or more `M_DATA` messages linked to it. These messages contain the data to be copied into user space.

The `copyresp`(9S) structure is contained in `M_IOCDATA` response message types. These messages are generated by the stream head in response to an `M_COPYIN` or `M_COPYOUT` request. If the message is in response to an `M_COPYOUT` request, the message has no messages attached to

it (`b_cont` is `NULL`). If the response is to an `M_COPYIN`, then zero or more `M_DATA` message types are attached to the `M_IOCDATA` message. These attached messages contain a copy of the user data requested by the `M_COPYIN` message.

The `iocblk`(9S), `copyreq`(9S), and `copyresp`(9S) structures contain a field indicating the type of `ioctl`(2) command, a pointer to the user's credentials, and a unique identifier for this `ioctl`(2). These fields must be preserved.

The structure member `cq_private` is reserved for use by the module. `M_COPYIN` and `M_COPYOUT` request messages contain a `cq_private` field that can be set to contain state information for `ioctl`(2) processing (which identifies what the subsequent `M_IOCDATA` response message contains). This state is returned in `cp_private` in the `M_IOCDATA` message. This state information determines the next step in processing the message. Keeping the state in the message makes the message self-describing and simplifies the `ioctl`(2) processing.

For each piece of data that the module copies from user space, an `M_COPYIN` message is sent to the stream head. The `M_COPYIN` message specifies the user address (`cq_addr`) and number of bytes (`cq_size`) to copy from user space. The stream head responds to the `M_COPYIN` request with a `M_IOCDATA` message. The `b_cont` field of the `M_IOCDATA` `mblk` contains the contents pointed to by the `M_COPYIN` request. Likewise, for each piece of data that the module copies to user space, an `M_COPYOUT` message is sent to the stream head. Specify the user address (`cq_addr`) and number of bytes to copy (`cq_size`). The data to be copied is linked to the `M_COPYOUT` message as one or more `M_DATA` messages. The stream head responds to `M_COPYOUT` requests with an `M_IOCDATA` message, but `b_cont` is null.

After the module has finished processing the `ioctl` (that is, all `M_COPYIN` and `M_COPYOUT` requests have been processed), the `ioctl`(2) must be acknowledged with an `M_IOCACK` to indicate successful completion of the command or an `M_IOCNAK` to indicate failure.

If an error occurs when attempting to copy data to or from user address space, the stream head will set `cp_rval` in the `M_IOCDATA` message to the error number. In the event of such an error, the `M_IOCDATA` message should be freed by the module or driver. No acknowledgement of the `ioctl`(2) is sent in this case.

Transparent `ioctl` Examples

Following are three examples of transparent `ioctl`(2) processing. Example 8-8 and Example 8-9 illustrate how to use `M_COPYIN` to copy data from user space. Example 8-10 illustrates how to use `M_COPYOUT` to copy data to user space. Example 8-11 is a more complex example showing state transitions that combine `M_COPYIN` and `M_COPYOUT`.

In these examples the message blocks are reused to avoid the overhead of allocating, copying, and releasing messages. This is standard practice.

The stream head guarantees that the size of the message block containing an `iocblk`(9S) structure is large enough to also hold the `copyreq`(9S) and `copyresp`(9S) structures.

`M_COPYIN` Example

Note - Please see the copyin section in "Writing Device Drivers for Oracle Solaris 11.2 " for information on the 64–bit data structure macros.

Example 8-8 illustrates the processing of a transparent `ioctl`(2) request only (nontransparent request processing is not shown). In this example, the contents of a user buffer are to be transferred into the kernel as part of an ioctl call of the form

```
ioctl(fd, SET_ADDR, (caddr_t) &bufadd);
```

where bufadd is a *struct address* whose elements are:

```
struct address {
    int   ad_len;;  /* buffer length in bytes */
    caddr_t  ad_addr;  /* buffer address */
};
```

This requires two pairs of messages (request and response) following receipt of the M_IOCTL message: the first `copyin`(9F), shown in Example 8-8, copies the structure (`address`) , and the second `copyin`(9F), shown in Example 8-9, copies the buffer (`address.ad.addr`). Two states are maintained and processed in this example: GETSTRUCT is for copying the address structure and GETADDR for copying the ad_addr of the structure.

The transparent part of the SET_ADDR M_IOCTL message processing requires that the address structure be copied from user address space. To accomplish this, the M_IOCTL message processing issues an M_COPYIN request to the stream head.

EXAMPLE 8-8   M_COPYIN: Copy the `address` Structure

```
struct address {  /* same members as in user space */
 int  ad_len; /* length in bytes */
 caddr_t ad_addr; /* buffer address */
};

/* state values (overloaded in private field) */
#define GETSTRUCT  0  /* address structure */
#define GETADDR  1  /* byte string from ad_addr */

static void xxioc(queue_t *q, mblk_t *mp);

static int
```

```
xxwput(q, mp)
 queue_t *q;  /* write queue */
 mblk_t *mp;
{
 struct iocblk *iocbp;
 struct copyreq *cqp;

 switch (mp->b_datap->db_type) {
  .
  .
  .
  case M_IOCTL:
   /* Process ioctl commands */
   iocbp = (struct iocblk *)mp->b_rptr;
   switch (iocbp->ioc_cmd) {
    case SET_ADDR;
     if (iocbp->ioc_count != TRANSPARENT) {
       /* do non-transparent processing here
        *       (not shown here) */
     } else {
      /* ioctl command is transparent
       * Reuse M_IOCTL block for first M_COPYIN
       * request of address structure */
      cqp = (struct copyreq *)mp->b_rptr;
      /* Get user space structure address from linked
       * M_DATA block */
      cqp->cq_addr = *(caddr_t *) mp->b_cont->b_rptr;
      cqp->cq_size = sizeof(struct address);
      /* MUST free linked blks */
      freemsg(mp->b_cont);
      mp->b_cont = NULL;

      /* identify response */
      cqp->cq_private = (mblk_t *)GETSTRUCT;

      /* Finish describing M_COPYIN message */
      cqp->cq_flag = 0;
      mp->b_datap->db_type = M_COPYIN;
      mp->b_wptr = mp->b_rptr + sizeof(struct copyreq);
      qreply(q, mp);
      break;
    default: /* M_IOCTL not for us */
      /* if module, pass on */
      /* if driver, nak ioctl */
      break;
   } /* switch (iocbp->ioc_cmd) */
   break;
  case M_IOCDATA:
   /* all M_IOCDATA processing done here */
   xxioc(q, mp);
   break;
 }
 return (0);
}
```

xxwput verifies that the SET_ADDR is TRANSPARENT to avoid confusion with an I_STR ioctl(2), which uses a value of ioc_cmd equivalent to the command argument of a transparent ioctl(2).

The `if else` statement checks whether the size count is equal to `TRANSPARENT`. If it is equal, the message was not generated from an `I_STR` [ioctl(2)](link) and the `else` clause of the `if else` executes.

```
if (iocbp->ioc_count != TRANSPARENT) {
    /* do non-transparent processing here (not shown here) */
} else {
```

The `mblk` is reused and mapped into a [copyreq(9S)](link) structure. The user space address of `bufadd` is contained in the `b_cont` of the `M_IOCTL` `mblk`. This address and its size are copied into the [copyreq(9S)](link) message. The `b_cont` of the copy request `mblk` is not needed, so it is freed and then filled with `NULL`.

```
cqp = (struct copyreq *)mp->b_rptr;
    /* Get user space structure address from linked M_DATA block */
    cqp->cq_addr = *(caddr_t *) mp->b_cont->b_rptr;
    cqp->cq_size = sizeof(struct address);
    /* MUST free linked blks */
    freemsg(mp->b_cont);
    mp->b_cont = NULL;
```

> ⚠️ **Caution -** The layout of the `iocblk`, `copyreq`, and `copyresp` structures is different between 32–bit and 64–bit kernels. Be careful not to overload any data structure in the `cp_private` or the `cq_filler` fields because alignment has changed.

EXAMPLE 8-9 `M_COPYIN`: Copy the Buffer Address

```
xxioc(queue_t *q, mblk_t *mp)   /* M_IOCDATA processing */
{
 struct iocblk *iocbp;
 struct copyreq *cqp;
 struct copyresp *csp;
 struct address *ap;

 csp = (struct copyresp *)mp->b_rptr;
 iocbp = (struct iocblk *)mp->b_rptr;

 /* validate this M_IOCDATA is for this module */
 switch (csp->cp_cmd) {
  case SET_ADDR:
   if (csp->cp_rval){ /* GETSTRUCT or GETADDR fail */
    freemsg(mp);
    return;
   }
   switch ((int)csp->cp_private){ /* determine state */
    case GETSTRUCT:  /* user structure has arrived */
      /* reuse M_IOCDATA block */
      mp->b_datap->db_type = M_COPYIN;
      mp->b_wptr = mp->b_rptr + sizeof (struct copyreq);
      cqp = (struct copyreq *)mp->b_rptr;
      /* user structure */
```

```
                 ap = (struct address *)mp->b_cont->b_rptr;
                 /* buffer length */
                 cqp->cq_size = ap->ad_len;
                 /* user space buffer address */
                 cqp->cq_addr = ap->ad_addr;
                 freemsg(mp->b_cont);
                 mp->b_cont = NULL;
                 cqp->cq_flag = 0;
                 cqp->cp_private=(mblk_t *)GETADDR;  /*nxt st*/
                 qreply(q, mp);
                 break;

            case GETADDR:    /* user address is here */
               /* hypothetical routine */
               if (xx_set_addr(mp->b_cont) == FAILURE) {
                mp->b_datap->db_type = M_IOCNAK;
                iocbp->ioc_error = EIO;
               } else {
                mp->b_datap->db_type=M_IOCACK;/*success*/
                /* can have been overwritten */
                iocbp->ioc_error = 0;
                iocbp->ioc_count = 0;
                iocbp->ioc_rval = 0;
               }
               mp->b_wptr=mp->b_rptr + sizeof (struct ioclk);
               freemsg(mp->b_cont);
               mp->b_cont = NULL;
               qreply(q, mp);
               break;

            default: /* invalid state: can't happen */
               freemsg(mp->b_cont);
               mp->b_cont = NULL;
               mp->b_datap->db_type = M_IOCNAK;
               mp->b_wptr = mp->rptr + sizeof(struct iocblk);
               /* can have been overwritten */
               iocbp->ioc_error = EINVAL;
               qreply(q, mp);
               break;
           }
           break;      /* switch (cp_private) */

        default: /* M_IOCDATA not for us */
         /* if module, pass message on */
         /* if driver, free message */
         break;
```

cq_private of the copy request is returned in cp_private of the copy response when the
M_IOCDATA message is returned. This value is set to GETSTRUCT to indicate that the address
structure is contained in the b_cont of the M_IOCDATA message mblk. The copy request message
is then sent back to the stream head. xxwput then returns and is called again when the stream
head responds with an M_IOCDATA message, which is processed by the xxioc routine

On receipt of the M_IOCDATA message for the SET_ADDR command, xxioc checks cp_rval. If an
error occurred during the copyin operation, cp_rval is set. The mblk is freed and, if necessary,

xxioc cleans up from previous `M_IOCTL` requests, freeing memory, resetting state variables, and so on. The stream head returns the appropriate error to the user.

```
if (csp->cp_rval){ /* GETSTRUCT or GETADDR fail */
  freemsg(mp);
  return;
```

If no error occurred during the `copyin` operation, the `switch` statement determines whether to process the user structure, GETSTRUCT, or user address, GETADDR.

```
switch ((int)csp->cp_private){  /*determine state*/
```

The `cp_private` field set to GETSTRUCT indicates that the linked `b_cont mblk` contains a copy of the user's `address` structure. The example then copies the actual address specified in `address.ad_addr`. The program issues another `M_COPYIN` request to the stream head, but this time `cq_private` contains GETADDR to indicate that the `M_IOCDATA` response will contain a copy of `address.ad_addr`. The stream head, copies the information at the requested user address, and sends it downstream in another, final `M_IOCDATA` message.

```
case GETSTRUCT:  /* user structure has arrived */
    /* reuse M_IOCDATA block */
    mp->b_datap->db_type = M_COPYIN;
    mp->b_wptr = mp->b_rptr + sizeof (struct copyreq);
    cqp = (struct copyreq *)mp->b_rptr;
    /* user structure */
    ap = (struct address *)mp->b_cont->b_rptr;
    /* buffer length */
    cqp->cq_size = ap->ad_len;
    /* user space buffer address */
    cqp->cq_addr = ap->ad_addr;
    freemsg(mp->b_cont);
    mp->b_cont = NULL;
    cqp->cq_flag = 0;
    cqp->cp_private=(mblk_t *)GETADDR;  /*nxt st*/
    qreply(q, mp);
    break;
```

The final `M_IOCDATA` message arrives from the stream head. `cp_private` contains GETADDR. The `ad_addr` data is contained in the `b_cont` link of the `mblk`. If the address is successfully processed by `xx_set_addr` (not shown here), the message is acknowledged with an `M_IOCACK` message. If `xx_set_addr` fails, the message is rejected with an `M_IOCNAK` message, `xx_set_addr` processes the user address from the `ioctl`(2).

After the final `M_IOCDATA` message is processed, the module acknowledges the `ioctl`(2) to let the stream head know that processing is complete. This is done by sending an `M_IOCACK` message upstream if the request was successfully processed. Always set `ioc_error` to zero, otherwise an error code could be passed to the user application. Set `ioc_rval` and `ioc_count` to zero to reflect that a return value of 0 and no data is to be passed upstream. If the request cannot be processed, either an `M_IOCNAK` or `M_IOCACK` can be sent upstream with an appropriate error number. When sending an `M_IOCNAK` or `M_IOCACK`, freeing the linked `M_DATA` block is not mandatory. It is more efficient to use the stream head handle to free the linked `M_DATA` block.

If ioc_error is set in an M_IOCNAK or M_IOCNACK message, this error code will be returned to the user. If no error code is set in an M_IOCNAK message, EINVAL will be returned to the user.

```
            case GETADDR:   /* user address is here */
/* hypothetical routine */
if (xx_set_addr(mp->b_cont) == FAILURE) {
 mp->b_datap->db_type = M_IOCNAK;
 iocbp->ioc_error = EIO;
} else {
 mp->b_datap->db_type=M_IOCACK;/*success*/
 /* can have been overwritten */
 iocbp->ioc_error = 0;
 iocbp->ioc_count = 0;
 iocbp->ioc_rval = 0;
}
mp->b_wptr=mp->b_rptr + sizeof (struct ioclk);
freemsg(mp->b_cont);
mp->b_cont = NULL;
qreply(q, mp);
break;
```

M_COPYOUT Example

Note - Please see the copyout section in "Writing Device Drivers for Oracle Solaris 11.2 " for information on the 64–bit data structure macros.

The following code excerpts return option values for the STREAMS device by placing them in the user's options structure. This is done by a transparent ioctl(2) call of the form

```
struct options optadd;

ioctl(fd, GET_OPTIONS,(caddr_t) &optadd)
```

or by a nontransparent I_STR call

```
 struct strioctl opts_strioctl;
 structure options optadd;

 opts_strioctl.ic_cmd = GET_OPTIONS;
 opts_strioctl.ic_timeout = -1
 opts_strioctl.ic_len = sizeof (struct options);
 opts_strioctl.ic_dp = (char *)&optadd;
 ioctl(fd, I_STR, (caddr_t) &opts_strioctl)
```

In the nontransparent I_STR case, opts_strioctl.ic_dp points to the options structure optadd.

Example 8-7 illustrates support of both the I_STR and transparent forms of ioctl(2). The transparent form requires a single M_COPYOUT message following receipt of the M_IOCTL to copy out the contents of the structure. xxwput is the write-side put procedure of module or driver xx.

EXAMPLE 8-10 M_COPYOUT

```
struct options {   /* same members as in user space */
 int   op_one;
 int   op_two;
 short   op_three;
 long   op_four;
};

static int
xxwput (queue_t *q, mblk_t *mp)
{
 struct iocblk *iocbp;
 struct copyreq *cqp;
 struct copyresp *csp;
 int transparent = 0;

 switch (mp->b_datap->db_type) {
  .
  .
  .
  case M_IOCTL:
   iocbp = (struct iocblk *)mp->b_rptr;
   switch (iocbp->ioc_cmd) {
    case GET_OPTIONS:
     if (iocbp->ioc_count == TRANSPARENT) {
       transparent = 1;
       cqp = (struct copyreq *)mp->b_rptr;
       cqp->cq_size = sizeof(struct options);
       /* Get struct address from
       linked M_DATA block */
       cqp->cq_addr = (caddr_t)
       *(caddr_t *)mp->b_cont->b_rptr;
       cqp->cq_flag = 0;
       /* No state necessary - we will only ever
      * get one M_IOCDATA from the Stream head
      * indicating success or failure for
      * the copyout */
      }
      if (mp->b_cont)
       freemsg(mp->b_cont);
      if ((mp->b_cont =
        allocb(sizeof(struct options),
          BPRI_MED)) == NULL) {
       mp->b_datap->db_type = M_IOCNAK;
       iocbp->ioc_error = EAGAIN;
       qreply(q, mp);
       break;
      }
      /* hypothetical routine */
      xx_get_options(mp->b_cont);
      if (transparent) {
        mp->b_datap->db_type = M_COPYOUT;
        mp->b_wptr = mp->b_rptr + sizeof(struct copyreq);
      } else {
        mp->b_datap->db_type = M_IOCACK;
```

```
        iocbp->ioc_count = sizeof(struct options);
      }
      qreply(q, mp);
      break;

     default: /* M_IOCTL not for us */
      /*if module, pass on;if driver, nak ioctl*/
      break;
    } /* switch (iocbp->ioc_cmd) */
    break;

   case M_IOCDATA:
    csp = (struct copyresp *)mp->b_rptr;
    /* M_IOCDATA not for us */
    if (csp->cmd != GET_OPTIONS) {
     /*if module/pass on, if driver/free message*/
     break;
    }
    if ( csp->cp_rval ) {
     freemsg(mp); /* failure */
     return (0);
    }
    /* Data successfully copied out, ack */

    /* reuse M_IOCDATA for ack */
    mp->b_datap->db_type = M_IOCACK;
    mp->b_wptr = mp->b_rptr + sizeof(struct iocblk);
    /* can have been overwritten */
    iocbp->ioc_error = 0;
    iocbp->ioc_count = 0;
    iocbp->ioc_rval = 0;
    qreply(q, mp);
    break;
    .
    .
    .
   } /* switch (mp->b_datap->db_type) */
   return (0);
```

xxwput first checks whether the ioctl(2) command is transparent. If it is, the message is
reused as an M_COPYOUT copy request message. The pointer to the receiving buffer is in the
linked message and is copied into cq_addr. Because only a single copy out is being done, no
state information needs to be stored in cq_private. The original linked message is freed, in
case it isn't big enough to hold the request.

```
if (iocbp->ioc_count == TRANSPARENT) {
  transparent = 1;
  cqp = (struct copyreq *)mp->b_rptr;
  cqp->cq_size = sizeof(struct options);
  /* Get struct address from linked M_DATA block */
  cqp->cq_addr = (caddr_t)
        *(caddr_t *)mp->b_cont->b_rptr;
  cqp->cq_flag = 0;
  /* No state necessary - we will only ever get one
   * M_IOCDATA from the Stream head indicating
   * success or failure for the copyout */
```

```
  }
 if (mp->b_cont)
  freemsg(mp->b_cont);
```

As an optimization, the following code checks the size of the message for reuse:

```
mp->b_cont->b_datap->db_lim
- mp->b_cont->b_datap->db_base >= sizeof (struct options)
```

Note - *Hardening Information*. After message reuse, make sure to retain the relation:

```
db_base <= b_rptr <= b_wptr <= db_lim
```

A new linked message is allocated to hold the option request. When using the transparent
ioctl(2) M_COPYOUT command, data contained in the linked message is passed to the stream
head. The stream head will copy the data to the user's address space and issue an M_IOCDATA
in response to the M_COPYOUT message, which the module must acknowledge in an M_IOCACK
message.

```
          /* hypothetical routine */
   xx_get_options(mp->b_cont);
   if (transparent) {
    mp->b_datap->db_type = M_COPYOUT;
    mp->b_wptr = mp->b_rptr + sizeof(struct copyreq);
   } else {
    mp->b_datap->db_type = M_IOCACK;
    iocbp->ioc_count = sizeof(struct options);
   }
```

If the message is not transparent (is issued through an I_STR ioctl(2)), the data is sent with
the M_IOCACK acknowledgement message and copied into the buffer specified by the strioctl
data structure. ioc_error, ioc_count, and ioc_rval are cleared to prevent any stale data from
being passed back to the stream head.

```
          /* reuse M_IOCDATA for ack */
   mp->b_datap->db_type = M_IOCACK;
   mp->b_wptr = mp->b_rptr + sizeof(struct iocblk);
   /* can have been overwritten */
   iocbp->ioc_error = 0;
   iocbp->ioc_count = 0;
   iocbp->ioc_rval = 0;
   qreply(q, mp);
   break;
```

Bidirectional Data Transfer Example

Example 8-11 illustrates bidirectional data transfer between the kernel and application during
transparent ioctl(2) processing. It also shows how to use more complex state information.

The user wants to send and receive data from user buffers as part of a transparent ioctl(2) call of the form:

```
 ioctl(fd, XX_IOCTL, (caddr_t) &addr_xxdata)
```

EXAMPLE 8-11   Bidirectional Data Transfer

```
struct xxdata {                /* same members in user space */
   int        x_inlen;     /* number of bytes copied in */
   caddr_t    x_inaddr;    /* buf addr of data copied in */
   int        x_outlen;    /* number of bytes copied out */
   caddr_t    x_outaddr;   /* buf addr of data copied out */
};
/* State information for ioctl processing */
struct state {
  int         st_state;    /* see below */
  struct xxdata  st_data;  /* see above */
};
/* state values */

#define GETSTRUC    0   /* get xxdata structure */
#define GETINDATA   1   /* get data from x_inaddr */
#define PUTOUTDATA  2   /* get response from M_COPYOUT */

static void xxioc(queue_t *q, mblk_t *mp);

static int
xxwput (queue_t *q,  mblk_t *mp) {
  struct iocblk *iocbp;
  struct copyreq *cqp;
  struct state *stp;
  mblk_t *tmp;

  switch (mp->b_datap->db_type) {
   .
   .
   .
   case M_IOCTL:
    iocbp = (struct iocblk *)mp->b_rptr;
    switch (iocbp->ioc_cmd) {
    case XX_IOCTL:
    /* do non-transparent processing. (See I_STR ioctl
     * processing discussed in previous section.)
     */
    /*Reuse M_IOCTL block for M_COPYIN request*/

    cqp = (struct copyreq *)mp->b_rptr;

    /* Get structure's user address from
     * linked M_DATA block */

    cqp->cq_addr = (caddr_t)
     *(long *)mp->b_cont->b_rptr;
    freemsg(mp->b_cont);
    mp->b_cont = NULL;
```

```
    /* Allocate state buffer */

    if ((tmp = allocb(sizeof(struct state),
     BPRI_MED)) == NULL) {
      mp->b_datap->db_type = M_IOCNAK;
      iocbp->ioc_error = EAGAIN;
      qreply(q, mp);
      break;
    }
    tmp->b_wptr += sizeof(struct state);
    stp = (struct state *)tmp->b_rptr;
    stp->st_state = GETSTRUCT;
    cqp->cq_private = tmp;

    /* Finish describing M_COPYIN message */

    cqp->cq_size = sizeof(struct xxdata);
    cqp->cq_flag = 0;
    mp->b_datap->db_type = M_COPYIN;
    mp->b_wptr=mp->b_rptr+sizeof(struct copyreq);
    qreply(q, mp);
    break;

  default: /* M_IOCTL not for us */
    /* if module, pass on */
    /* if driver, nak ioctl */
    break;

  } /* switch (iocbp->ioc_cmd) */
  break;

 case M_IOCDATA:
  xxioc(q, mp);    /*all M_IOCDATA processing here*/
  break;
  .
  .
  .
 } /* switch (mp->b_datap->db_type) */
}
```

Three pairs of messages are required following the `M_IOCTL` message:

1. case `GETSTRUCT` copies the structure into the message buffer.
2. case `GETINDATA` copies the user buffer into the message buffer.
3. case `PUTOUTDATA` copies the second message buffer into the user buffer.

xxwput is the write-side `put` procedure for module or driver xx. xxwput allocates a message block to contain the state structure and reuses the `M_IOCTL` to create an `M_COPYIN` message to read in the xxdata structure.

`M_IOCDATA` processing is done in xxioc as shown in the following example:

EXAMPLE 8-12 M_IOCDATA Processing

```
xxioc(     /* M_IOCDATA processing */
 queue_t *q,
 mblk_t *mp)
{
 struct iocblk *iocbp;
 struct copyreq *cqp;
 struct copyresp *csp;
 struct state *stp;
 mblk_t *xx_indata();

 csp = (struct copyresp *)mp->b_rptr;
 iocbp = (struct iocblk *)mp->b_rptr;
 switch (csp->cp_cmd) {

 case XX_IOCTL:
   if (csp->cp_rval) { /* failure */
    if (csp->cp_private) /* state structure */
      freemsg(csp->cp_private);
    freemsg(mp);
    return;
    }
   stp = (struct state *)csp->cp_private->b_rptr;
   switch (stp->st_state) {

   case GETSTRUCT:   /* xxdata structure copied in */
         /* save structure */

    stp->st_data =
     *(struct xxdata *)mp->b_cont->b_rptr;
    freemsg(mp->b_cont);
    mp->b_cont = NULL;
    /* Reuse M_IOCDATA to copyin data */
    mp->b_datap->db_type = M_COPYIN;
    cqp = (struct copyreq *)mp->b_rptr;
    cqp->cq_size = stp->st_data.x_inlen;
    cqp->cq_addr = stp->st_data.x_inaddr;
    cqp->cq_flag = 0;
    stp->st_state = GETINDATA; /* next state */
    qreply(q, mp);
    break;

   case GETINDATA: /* data successfully copied in */
    /* Process input, return output */
    if ((mp->b_cont = xx_indata(mp->b_cont))
     == NULL) { /* hypothetical */
       /* fail xx_indata */
       mp->b_datap->db_type = M_IOCNAK;
       mp->b_wptr = mp->b_rptr +
        sizeof(struct iocblk);
      iocbp->ioc_error = EIO;
      qreply(q, mp);
      break;
     }
    mp->b_datap->db_type = M_COPYOUT;
    cqp = (struct copyreq *)mp->b_rptr;
```

```
    cqp->cq_size = min(msgdsize(mp->b_cont),
     stp->st_data.x_outlen);
    cqp->cq_addr = stp->st_data.x_outaddr;
    cqp->cq_flag = 0;
    stp->st_state = PUTOUTDATA; /* next state */
    qreply(q, mp);
    break;

  case PUTOUTDATA: /* data copied out, ack ioctl */
    freemsg(csp->cp_private); /*state structure*/
    mp->b_datap->db_type = M_IOCACK;
    mp->b_wtpr = mp->b_rptr + sizeof (struct iocblk);
            /* can have been overwritten */
    iocbp->ioc_error = 0;
    iocbp->ioc_count = 0;
    iocbp->ioc_rval = 0;
    qreply(q, mp);
    break;

  default: /* invalid state: can't happen */
    freemsg(mp->b_cont);
    mp->b_cont = NULL;
    mp->b_datap->db_type = M_IOCNAK;
    mp->b_wptr=mp->b_rptr + sizeof (struct iocblk);
    iocbp->ioc_error = EINVAL;
    qreply(q, mp);
    break;
  } /* switch (stp->st_state) */
  break;
 default: /* M_IOCDATA not for us */
  /* if module, pass message on */
  /* if driver, free message */
  break;
 } /* switch (csp->cp_cmd) */
}
```

At `case GETSTRUCT`, the user `xxdata` structure is copied into the module's state structure (pointed to by `cp_private` in the message) and the `M_IOCDATA` message is reused to create a second `M_COPYIN` message to read the user data.

At `case GETINDATA`, the input user data is processed by `xx_indata` (not supplied in the example), which frees the linked `M_DATA` block and returns the output data message block. The `M_IOCDATA` message is reused to create an `M_COPYOUT` message to write the user data.

At `case PUTOUTDATA`, the message block containing the state structure is freed and an acknowledgement is sent upstream.

Care must be taken at the "can't happen" `default` case since the message block containing the state structure (`cp_private`) is not returned to the pool because it might not be valid. This might result in a lost block. The `ASSERT` helps find errors in the module if a "can't happen" condition occurs.

I_LIST ioctl(2)Example

The I_LIST ioctl(2) lists the drivers and module in a stream.

EXAMPLE 8-13   List a Stream's Drivers and Modules

```
#include <stdio.h>
#include <string.h>
#include <stropts.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/socket.h>

main(int argc, const char **argv)
{
  int              s, i;
  int              mods;
  struct str_list   mod_list;
  struct str_mlist *mlist;

  /* Get a socket... */
  if((s = socket(AF_INET, SOCK_STREAM, 0)) <= 0) {
   perror("socket: ");
   exit(1);
  }

  /* Determine the number of modules in the stream. */
  if((mods = ioctl(s, I_LIST, 0)) < 0){
   perror("I_LIST ioctl");
  }
  if(mods == 0) {
   printf("No modules\n");
   exit(1);
  } else {
   printf("%d modules\n", mods);
  }
  /* Allocate memory for all of the module names... */
  mlist = (struct str_mlist *) calloc(mods, sizeof(struct str_mlist));
  if(mlist == 0){
   perror("malloc failure");
   exit(1);
  }
  mod_list.sl_modlist = mlist;
  mod_list.sl_nmods = mods;

  /* Do the ioctl and get the module names. */
  if(ioctl(s, I_LIST, &mod_list) < 0){
   perror("I_LIST ioctl fetch");
   exit(1);
  }

  /* Print out the name of the modules */
  for(i = 0; i < mods; i++) {
   printf("s: %s\n", mod_list.sl_modlist[i].l_name);
```

```
  }

  free(mlist);

  exit(0);
}
```

M_FLUSH Message Handling

All modules and drivers are expected to handle M_FLUSH messages. An M_FLUSH message can originate at the stream head or from a module or a driver. The user can cause data to be flushed from queued messages of a stream by submitting an I_FLUSH ioctl(2). Data can be flushed from the read side, write side, or both sides of a stream.

```
ioctl(fd,I_FLUSH, arg);
```

The first byte of the M_FLUSH message is an option flag. The following table describes the possible values for this flag.

TABLE 8-1   M_FLUSH Arguments and bi_flag Values

Flag	Description
FLUSHR	Flush read side of stream
FLUSHW	Flush write queue
FLUSHRW	Flush both, read and write, queues
FLUSHBAND	Flush a specified priority band only

Flushing According to Priority Bands

In addition to being able to flush all the data from a queue, a specific band can be flushed using the I_FLUSHBAND ioctl(2).

```
ioctl(fd, I_FLUSHBAND, bandp);
```

The ioctl(2) is passed a pointer to a bandinfo structure. The bi_pri field indicates the band priority to be flushed (from 0 to 255). The bi_flag field indicates the type of flushing to be done. The legal values for bi_flag are defined in Table 8-1. bandinfo has the following format:

```
struct bandinfo {
  unsigned char      bi_pri;
  in                 bi_flag;
```

```
};
```

See "`M_FLUSH`" on page 294 for details on how modules and drivers should handle flush band requests.

Figure 8-1 and Figure 8-2 further demonstrate flushing the entire stream due to a line break. Figure 8-1 shows the flushing of the write side of a stream, and Figure 8-2 shows the flushing of the read side of a stream.

FIGURE   8-1   Flushing the Write Side of a Stream
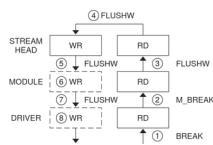
The following discussion describes the sequence of events shown in Figure 8-1 (dotted lines mean flushed queues):

1. A break is detected by a driver.
2. The driver generates an `M_BREAK` message and sends it upstream.
3. The module translates the `M_BREAK` into an `M_FLUSH` message with `FLUSHW` set, then sends it upstream.
4. The stream head does *not* flush the write queue (no messages are ever queued there).
5. The stream head turns the message around (sends it down the write side).
6. The module flushes its write queue.
7. The message is passed downstream.
8. The driver flushes its write queue and frees the message.

Figure 8-2 shows flushing the read side of a stream.

FIGURE   8-2   Flushing the Read Side of a Stream
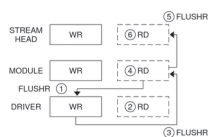
The following discussion describes the sequence of events.

1. After generating the first M_FLUSH message, the module generates an M_FLUSH with FLUSHR set and sends it downstream.
2. The driver flushes its read queue.
3. The driver turns the message around (sends it up the read side).
4. The module flushes its read queue.
5. The message is passed upstream.
6. The stream head flushes the read queue and frees the message.

The following code shows line discipline module for flush handling.

EXAMPLE 8-14 Line Discipline Module for Flush Handling

```
static int
ld_put(
  queue_t *q,     /* pointer to read/write queue */
  mblk_t *mp)     /* pointer to message being passed */
{
  switch (mp->b_datap->db_type) {
   default:
    putq(q, mp); /* queue everything */
   return (0);      /* except flush */

   case M_FLUSH:
    if (*mp->b_rptr & FLUSHW)      /* flush write q */
      flushq(WR(q), FLUSHDATA);

    if (*mp->b_rptr & FLUSHR)      /* flush read q */
      flushq(RD(q), FLUSHDATA);

    putnext(q, mp);        /* pass it on */
    return(0);
  }
}
```

The above example uses FLUSHDATA as the argument to flushq. This code will flush data type messages (M_DATA, M_DELAY, M_PROTO and M_PCPROTO) only. Flushing non-data messages, such as M_IOCTL, M_IOCACK, and M_IOCNAK, might flush messages that contain critical state in the stream. If an M_IOCACK message is flushed, a thread waiting at the stream head for a non-data message response will never receive it. Care should be taken when using flushq.

The stream head turns around the M_FLUSH message if FLUSHW is set (FLUSHR is cleared). A driver turns around M_FLUSH if FLUSHR is set (should mask off FLUSHW).

Flushing Priority Band

The bi_flag field is one of FLUSHR, FLUSHW, or FLUSHRW.

The following example shows flushing according to the priority band.

EXAMPLE 8-15 Priority Band Data Flush Handling

```
queue_t *rdq;          /* read queue */
queue_t *wrq;          /* write queue */

 case M_FLUSH:
  if (*bp->b_rptr & FLUSHBAND) {
   if (*bp->b_rptr & FLUSHW)
    flushband(wrq, FLUSHDATA, *(bp->b_rptr + 1));
   if (*bp->b_rptr & FLUSHR)
    flushband(rdq, FLUSHDATA, *(bp->b_rptr + 1));
  } else {
   if (*bp->b_rptr & FLUSHW)
    flushq(wrq, FLUSHDATA);
   if (*bp->b_rptr & FLUSHR)
    flushq(rdq, FLUSHDATA);
  }
  /*
   * modules pass the message on;
   * drivers shut off FLUSHW and loop the message
   * up the read-side if FLUSHR is set; otherwise,
   * drivers free the message.
   */
  break;
```

Note that modules and drivers are not required to treat messages as flowing in separate bands. Modules and drivers can view the queue as having only two bands of flow, normal and high priority. However, the latter alternative flushes the entire queue whenever an M_FLUSH message is received.

The field b_flag of the msgb structure provides a way for the stream head to stop M_FLUSH messages from being reflected forever when the stream is used as a pipe. When the stream head receives an M_FLUSH message, it sets the MSGNOLOOP flag in the b_flag field before reflecting the message down the write side of the stream. If the stream head receives an M_FLUSH message with this flag set, the message is freed rather than reflected.

FIGURE 8-3 Interfaces Affecting Drivers
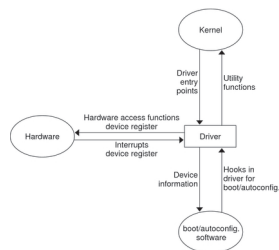
The set of STREAMS utilities available to drivers are listed in Appendix B, "Kernel Utility Interface Summary". No system-defined macros that manipulate global kernel data or introduce structure-size dependencies are permitted in these utilities. So, some utilities that have been implemented as *macros* in the prior Oracle Solaris operating environment releases are implemented as *functions* in the SunOS 5 system. This does not preclude the existence of both *macro* and *function* versions of these utilities. Driver source code should include a header file that picks up *function* declarations while the core operating system source should include a header file that defines the *macros*. With the DKI interface, the following STREAMS utilities are implemented as C programming language functions: `datamsg`(9F), `OTHERQ`(9F), `putnext`(9F), `RD`(9F), and `WR`(9F).

Replacing macros such as RD with function equivalents in the driver source code allows driver objects to be insulated from changes in the data structures and their size, increasing the useful lifetime of driver source code and objects. Multithreaded drivers are also protected against changes in implementation-specific STREAMS synchronization.

The DKI defines an interface suitable for drivers and there is no need for drivers to access global kernel data structures directly. The kernel function `drv_getparm`(9F) fetches information from these structures. This restriction has an important consequence. Because drivers are not permitted to access global kernel data structures directly, changes in the contents/offsets of information within these structures will not break objects.

Driver and Module Service Interfaces

STREAMS provides the means to implement a service interface between any two components in a stream, and between a user process and the topmost module in the stream. A *service interface* is a set of primitives defined at the boundary between a service user and a service provider (see Figure 8-5). Rules define a service and the allowable state transitions that result as these primitives are passed between the user and the provider. These rules are typically represented by a state machine. In STREAMS, the service user and provider are implemented in a module, driver, or user process. The primitives are carried bidirectionally between a service user and provider in `M_PROTO` and `M_PCPROTO` messages.

PROTO messages (`M_PROTO` and `M_PCPROTO`) can be multiblock. The second through last blocks are of type `M_DATA`. The first block in a PROTO message contains the control part of the primitive in a form agreed upon by the user and provider. The block is not intended to carry protocol headers. (Upstream PROTO messages can have multiple PROTO blocks at the start of the message, although its use is not recommended. `getmsg`(2) compacts the blocks into a single control part when sending to a user process.) The `M_DATA` block contains any data part associated with the primitive. The data part can be processed in a module that receives it, or it can be sent to the next stream component, along with any data generated by the module. The contents of `PROTO` messages and their allowable sequences are determined by the service interface specification.

PROTO messages can be sent bidirectionally (upstream and downstream) on a stream and between a stream and a user process. putmsg(2) and getmsg(2) system calls are analogous to write(2) and read(2) except that the former allow both data and control parts to be (separately) passed, and they retain the message boundaries across the user-stream interface. putmsg(2) and getmsg(2) separately copy the control part (M_PROTO or M_PCPROTO block) and data part (M_DATA blocks) between the stream and user process.

An M_PCPROTO message normally is used to acknowledge primitives composed of other messages. M_PCPROTO ensures that the acknowledgement reaches the service user before any other message. If the service user is a user process, the stream head will only store a single M_PCPROTO message, and discard subsequent M_PCPROTO messages until the first one is read with getmsg(2).

FIGURE   8-4   Protocol Substitution
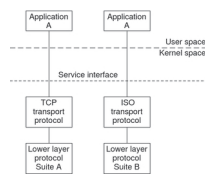
By defining a service interface through which applications interact with a transport protocol, you can substitute a different protocol below the service interface that is completely transparent to the application. In Figure 8-4, the same application can run over the Transmission Control Protocol (TCP) and the ISO transport protocol. Of course, the service interface must define a set of services common to both protocols.

The three components of any service interface are the service user, the service provider, and the service interface itself, as seen in Figure 8-5.

FIGURE   8-5   Service Interface
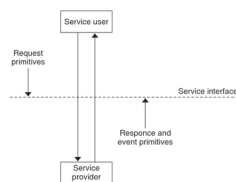
Typically, an application makes requests of a service provider using some well-defined service primitive. Responses and event indications are also passed from the provider to the user using service primitives.

Each service interface primitive is a distinct STREAMS message that has two parts, a control part and a data part. The control part contains information that identifies the primitive and includes all necessary parameters. The data part contains user data associated with that primitive.

An example of a service interface primitive is a transport protocol connect request. This primitive requests the transport protocol service provider to establish a connection with another transport user. The parameters associated with this primitive can include a destination protocol address and specific protocol options to be associated with that connection. Some transport protocols also allow a user to send data with the connect request. A STREAMS message is used to define this primitive. The control part identifies the primitive as a connect request and includes the protocol address and options. The data part contains the associated user data.

Service Interface Library Example

The service interface library example presented here includes four functions that enable a user do the following:

- Establish a stream to the service provider and bind a protocol address to the stream
- Send data to a remote user
- Receive data from a remote user
- Close the stream connected to the provider

First, the structure and constant definitions required by the library are shown in the following code. These typically reside in a header file associated with the service interface.

The defined structures describe the contents of the control part of each service interface message passed between the service user and service provider. The first field of each control part defines the type of primitive being passed.

EXAMPLE 8-16 Service Interface Library Header File

```
/*
 * Primitives initiated by the service user.
 */
#define BIND_REQ                1   /* bind request */
#define UNITDATA_REQ            2   /* unitdata request */

/*
 * Primitives initiated by the service provider.
 */
#define OK_ACK                  3   /* bind acknowledgement */
#define ERROR_ACK               4   /* error acknowledgement */
#define UNITDATA_IND            5   /* unitdata indication */
```

```
/*
 * The following structure definitions define the format
 * of the control part of the service interface message
 * of the above primitives.
 */
struct bind_req {                       /* bind request */
  t_scalar_t   PRIM_type;               /* always BIND_REQ */
  t_uscalar_t  BIND_addr;               /* addr to bind */
};
 struct unitdata_req {                  /* unitdata request */
  t_scalar_t   PRIM_type;               /* always UNITDATA_REQ */
  t_scalar_t   DEST_addr;               /* destination addr */
};

struct ok_ack {                         /* positiv acknowledgement*/
  t_scalar_t   PRIM_type;               /* always OK_ACK */
};

struct error_ack {                      /* error acknowledgement */
  t_scalar_t   PRIM_type;               /* always ERROR_ACK */
  t_scalar_t   UNIX_error;              /* UNIX systemerror code */
};

struct unitdata_ind {                   /* unitdata indication */
  t_scalar_t   PRIM_type;               /* always UNITDATA_IND */
  t_scalar_t   SRC_addr;                /* source addr */
};

/* union of all primitives */
union primitives {
  t_scalar_t            type;
  struct bind_req       bind_req;
  struct unitdata_req   unitdata_req;
  struct ok_ack         ok_ack;
  struct error_ack      error_ack;
  struct unitdata_ind   unitdata_ind;
};

/* header files needed by library */
#include <stropts.h>
#include <stdio.h>
#include <errno.h>
```

Five primitives are defined. The first two represent requests from the service user to the service provider.

BIND_REQ   Asks the provider to bind a specified protocol address. It requires an acknowledgement from the provider to verify that the contents of the request were syntactically correct.

UNITDATA_REQ   Asks the provider to send data to the specified destination address. It does not require an acknowledgement from the provider.

The three other primitives represent acknowledgements of requests, or indications of incoming events, and are passed from the service provider to the service user.

OK_ACK   Informs the user that a previous bind request was received successfully
                            by the service provider.

ERROR_ACK   Informs the user that a nonfatal error was found in the previous bind
                            request. It indicates that no action was taken with the primitive that
                            caused the error.

UNITDATA_IND   Indicates that data destined for the user has arrived.

Module Service Interface Example

The following code is part of a module that illustrates the concept of a service interface. The
module implements a simple service interface and mirrors the service interface library example.
The following rules pertain to service interfaces.

- Modules and drivers that support a service interface must act upon *all* PROTO messages
  and not pass them through.
- Modules can be inserted between a service user and a service provider to manipulate the
  data part as it passes between them. However, these modules cannot alter the contents of
  the control part (PROTO block, first message block) nor alter the boundaries of the control
  or data parts. That is, the message blocks comprising the data part can be changed, but the
  message cannot be split into separate messages nor combined with other messages.

  In addition, modules and drivers must observe the rule that high-priority messages are not
  subject to flow control and forward them accordingly.

Service Primitive Declarations

The service interface primitives are defined in the declarations shown in the following example:

EXAMPLE 8-17   Service Primitive Declarations

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/stream.h>
#include <sys/errno.h>

/* Primitives initiated by the service user */

#define BIND_REQ    1  /* bind request */
#define UNITDATA_REQ   2  /* unitdata request */

 /* Primitives initiated by the service provider */

#define OK_ACK    3  /* bind acknowledgement */
#define ERROR_ACK    4  /* error acknowledgement */
#define UNITDATA_IND   5  /* unitdata indication */
/*
```

```
 * The following structures define the format of the
 * stream message block of the above primitives.

 */
struct bind_req {                    /* bind request */
  t_scalar_t   PRIM_type;           /* always BIND_REQ */
  t_uscalar_t  BIND_addr;           /* addr to bind */
};
struct unitdata_req {                /* unitdata request */
  t_scalar_t   PRIM_type;           /* always UNITDATA_REQ */
  t_scalar_t   DEST_addr;           /* dest addr */
};
struct ok_ack {                      /* ok acknowledgement */
  t_scalar_t   PRIM_type;           /* always OK_ACK */
};
struct error_ack {                   /* error acknowledgement */
  t_scalar_t   PRIM_type;           /* always ERROR_ACK */
  t_scalar_t   UNIX_error;          /* UNIX system error code*/
};
struct unitdata_ind {                /* unitdata indication */
  t_scalar_t   PRIM_type;           /* always UNITDATA_IND */
  t_scalar_t   SRC_addr;            /* source addr */
};

union primitives {      /* union of all primitives */
  long                 type;
  struct bind_req      bind_req;
  struct unitdata_req  unitdata_req;
  struct ok_ack        ok_ack;
  struct error_ack     error_ack;
  struct unitdata_ind  unitdata_ind;
};
struct dgproto {                      /* structure minor device */
  short state;                        /* current provider state */
  long addr;                          /* net address */
};

/* Provider states */
#define IDLE 0
#define BOUND 1
```

In general, the `M_PROTO` or `M_PCPROTO` block is described by a data structure containing the service interface information in this example, union primitives.

The module recognizes two commands:

BIND_REQ   Give this stream a protocol address (for example, give it a name on the
                    network). After a `BIND_REQ` is completed, data from other senders will
                    find their way through the network to this particular stream.

UNITDATA_REQ   Send data to the specified address.

The module generates three messages:

OK_ACK   A positive acknowledgement (`ack`) of `BIND_REQ`.

ERROR_ACK   A negative acknowledgement (nak) of BIND_REQ.

UNITDATA_IND   Data from the network has been received.

The acknowledgement of a BIND_REQ informs the user that the request was syntactically correct (or incorrect if ERROR_ACK). The receipt of a BIND_REQ is acknowledged with an M_PCPROTO to ensure that the acknowledgement reaches the user before any other message. For example, if a UNITDATA_IND comes through before the bind is completed, the application cannot send data to the proper address

The driver uses a per-minor device data structure, dgproto, which contains the following:

state   Current state of the service provider IDLE or BOUND.

addr   Network address that has been bound to this stream.

The module open procedure is assumed to have set the write queue q_ptr to point at the appropriate private data structure, although this is not shown explicitly.

Service Interface Procedure

The write put procedure, protowput, is shown in the following example:

EXAMPLE 8-18   Service Interface protoput Procedure

```
static int protowput(queue_t *q, mblk_t *mp)
{
  union primitives *proto;
  struct dgproto *dgproto;
  int err;
  dgproto = (struct dgproto *) q->q_ptr;  /* priv data struct */
  switch (mp->b_datap->db_type) {
  default:
    /* don't understand it */
    mp->b_datap->db_type = M_ERROR;
    mp->b_rptr = mp->b_wptr = mp->b_datap->db_base;
    *mp->b_wptr++ = EPROTO;
    qreply(q, mp);
    break;
  case M_FLUSH: /* standard flush handling goes here ... */
    break;
  case M_PROTO:
    /* Protocol message -> user request */
    proto = (union primitives *) mp->b_rptr;
    switch (proto->type) {
    default:
     mp->b_datap->db_type = M_ERROR;
     mp->b_rptr = mp->b_wptr = mp->b_datap->db_base;
     *mp->b_wptr++ = EPROTO;
     qreply(q, mp);
     return;
```

```
          case BIND_REQ:
           if (dgproto->state != IDLE) {
             err = EINVAL;
             goto error_ack;
           }
           if (mp->b_wptr - mp->b_rptr !=
            sizeof(struct bind_req)) {
             err = EINVAL;
             goto error_ack;
           }
           if (err = chkaddr(proto->bind_req.BIND_addr))
             goto error_ack;
           dgproto->state = BOUND;
           dgproto->addr = proto->bind_req.BIND_addr;
           mp->b_datap->db_type = M_PCPROTO;
           proto->type = OK_ACK;
           mp->b_wptr=mp->b_rptr+sizeof(structok_ack);
           qreply(q, mp);
           break;
         error_ack:
           mp->b_datap->db_type = M_PCPROTO;
           proto->type = ERROR_ACK;
           proto->error_ack.UNIX_error = err;
           mp->b_wptr = mp->b_rptr+sizeof(structerror_ack);
           qreply(q, mp);
           break;
          case UNITDATA_REQ:
           if (dgproto->state != BOUND)
             goto bad;
           if (mp->b_wptr - mp->b_rptr !=
             sizeof(struct unitdata_req))
             goto bad;
           if(err=chkaddr(proto->unitdata_req.DEST_addr))
             goto bad;
           putq(q, mp);
           /* start device or mux output ... */
           break;
          bad:
           freemsg(mp);
           break;
          }
      }
return(0);
}
```

The write put procedure, protowput, switches on the message type. The only types accepted
are M_FLUSH and M_PROTO. For M_FLUSH messages, the driver performs the canonical flush
handling (not shown). For M_PROTO messages, the driver assumes that the message block
contains a union primitive and switches on the type field. Two types are understood: BIND_REQ
and UNITDATA_REQ.

For a BIND_REQ, the current state is checked; it must be IDLE. Next, the message size is checked.
If it is the correct size, the passed-in address is verified for legality by calling chkaddr. If
everything checks, the incoming message is converted into an OK_ACK and sent upstream. If
there was any error, the incoming message is converted into an ERROR_ACK and sent upstream.

For `UNITDATA_REQ`, the state is also checked; it must be `BOUND`. As above, the message size and destination address are checked. If there is any error, the message is discarded. If all is well, the message is put in the queue, and the lower half of the driver is started.

If the write `put` procedure receives a message type that it does not understand, either a bad `b_datap->db_type` or bad `proto->type`, the message is converted into an `M_ERROR` message and is then sent upstream.

The generation of `UNITDATA_IND` messages (not shown in the example) would normally occur in the device interrupt if this is a hardware driver or in the lower read `put` procedure if this is a multiplexer. The algorithm is simple: the data part of the message is prefixed by an `M_PROTO` message block that contains a `unitdata_ind` structure and is sent upstream.

Message Type Change Rules

To change a message type, use the following rules:

■ You can only change the `M_IOCTL` family of message types to other `M_IOCTL` message types.

■ `M_DATA`, `M_PROTO`, and `M_PCPROTO` are dependent on the modules, driver and service provider interfaces defined.

■ A message type should not be changed if the reference count > 1.

■ The data of a message should not be modified if the reference count > 1.

■ All other message types are interchangeable as long as sufficient space has been allocated in the data buffer of the message.

Common `ioctl` Interfaces

Many `ioctl` operations are common to a class of STREAMS drivers or STREAMS modules. Modules that deal with terminals usually implement a subset of the `termio`(7I) ioctls. Similarly, drivers that deal with audio devices usually implement a subset of the `audio`(7I) interfaces.

Because no data structures have changed size as a result of the LP64 data model for either `termio`(7I) or `audio`(7I), you do no need to use any of the structure macros to decode any of these `ioctls`.

FIORDCHK

The FIORDCHK ioctl returns a count (in bytes) of the number of bytes to be read as the return value. Although FIORDCHK should be able to return more than MAXINT bytes, it is constrained to returning an int by the type of the ioctl(2) function.

FIONREAD

The FIONREAD ioctl returns the number of data bytes (in all data messages queued) in the location pointed to by the arg parameter. The ioctl returns a 32-bit quantity for both 32-bit and 64-bit application., Therefore, code that passes the address of a long variable needs to be changed to pass an int variable for 64–bit applications.

I_NREAD

The I_NREAD ioctl (streamio(7I)) is an informational ioctl that counts the data bytes as well as the number of messages in the stream head read queue. The number of data bytes is returned in the location pointed to by the arg parameter of the ioctl. The number of messages in the stream head read queue is returned as the return value of the ioctl.

Like FIONREAD, the arg parameter to the I_NREAD ioctl should be a pointer to an int, not a long. And, like FIORDCHK, the return value is constrained to be less than or equal to MAXINT bytes, even if more data is available.

signal Message

STREAMS modules and drivers send signals to application processes through a special signal message. If the signal specified by the module or driver is not SIGPOLL (see signal(3C)), the signal is delivered to the process group associated with the stream. If the signal is SIGPOLL, the signal is only sent to processes that have registered for the signal by using the I_SETSIG ioctl(2).

Modules or drivers use an M_SIG message to insert an explicit in-band signal into a message stream. For example, a message can be sent to the application process immediately before a particular service interface message. When the M_SIG message reaches the head of the stream read queue, a signal is generated and the M_SIG message is removed. The service interface

message is the next message to be processed by the user. (The `M_SIG` message is usually defined as part of the service interface of the driver or module.)

♦♦♦ CHAPTER 9

9

STREAMS Drivers

This chapter describes the operation of STREAMS drivers and some of the processing typically required in the drivers.

- "STREAMS Device Drivers" on page 163
- "STREAMS Driver Entry Points" on page 164
- "STREAMS Driver Code Samples" on page 167

STREAMS Device Drivers

STREAMS drivers can be considered a subset of device drivers in general and character device drivers in particular. While there are some differences between STREAMS drivers and non-STREAMS drivers, much of the information contained in "Writing Device Drivers for Oracle Solaris 11.2 " also applies to STREAMS drivers.

Note - The word *module* is used differently when talking about drivers. A device driver is a kernel-loadable module that provides the interface between a device and the Device Driver Interface, and is linked to the kernel when it is first invoked.

STREAMS drivers share a basic programming model with STREAMS modules. Information common to both drivers and modules is discussed in Chapter 10, "STREAMS Modules". After summarizing some basic device driver concepts, this chapter discusses several topics specific to STREAMS device drivers (and not covered elsewhere) and then presents code samples illustrating basic STREAMS driver processing.

Basic Driver

A *device driver* is a loadable kernel module that translates between an I/O device and the kernel to operate the device.

Device drivers can also be software-only, implementing a pseudo-device such as RAM disk or a pseudo-terminal that only exists in software.

In the Oracle Solaris operating environment, the interface between the kernel and device drivers is called the Device Driver Interface (DDI/DKI). This interface is specified in the Section 9E manual pages that specify the driver entry points. Section 9 also details the kernel data structures (9S) and utility functions (9F) available to drivers.

The DDI protects the kernel from device specifics. Application programs and the rest of the kernel need little (if any) device-specific code to use the device. The DDI makes the system more portable and easier to maintain.

There are three basic types of device drivers corresponding to the three basic types of devices. *Character* devices handle data serially and transfer data to and from the processor one character at a time, the same as keyboards and low performance printers. *Serial block* devices and drivers also handle data serially, but transfer data to and from memory without processor intervention, the same as tape drives. *Direct access block* devices and drivers also transfer data without processor intervention and blocks of storage on the device can be addressed directly, the same as disk drives.

There are two types of character device drivers: standard character device drivers and STREAMS device drivers. STREAMS is a separate programming model for writing a character driver. Devices that receive data asynchronously (such as terminal and network devices) are suited to a STREAMS implementation.

STREAMS drivers share some kinds of processing with STREAMS modules. Important differences between drivers and modules include how the application manipulates drivers and modules and how interrupts are handled. In STREAMS, drivers are *opened* and modules are *pushed*. A device driver has an interrupt routine to process hardware interrupts.

STREAMS Driver Entry Points

STREAMS drivers have five different points of contact with the kernel:

TABLE 9-1   Kernel Contact Points

Kernel Contact Point	Description
Configuration entry points	These kernel (dynamically loading) routines enable the kernel to find the driver binary in the file system and load it into, or unload it from, the running kernel. The entry points include _init(9E), _info(9E), and _fini(9E).
Initialization entry points	Enable the driver to determine a device's presence and initialize its state. These routines are accessed through the dev_ops(9S) data structure during system initialization. They include getinfo(9E), probe(9E), attach(9E), and detach(9E).
Table-driven entry points	Accessed through cb_ops(9S), the character and block access table, when an application calls the appropriate interface. The members of the cb_ops(9S) structure include pointers to entry points that perform the device's functions,

Kernel Contact Point	Description
	such as `read`(9E), `write`(9E), `ioctl`(9E). The `cb_ops`(9S) table contains a pointer to the `streamtab`(9S)structure.
STREAMS queue processing entry points	Contained in the `streamtab`, read and process the STREAMS messages that travel through the queue structures. Examples of STREAMS queue processing entry points are `put`(9E) and `srv`(9E).
Interrupt routines	Handle the interrupts from the device (or software interrupts). It is added to the kernel by `ddi_add_intr`(9F) when the kernel configuration software calls `attach`(9E).

STREAMS Configuration Entry Points

As with other SunOS 5 drivers, STREAMS drivers are dynamically linked and loaded when referred to for the first time. For example, when the system is initially booted, the STREAMS pseudo-tty slave pseudo-driver (`pts`(7D)) is loaded automatically into the kernel when it is first opened.

In STREAMS, the header declarations differ between drivers and modules. The word "module" is used in two different ways when talking about drivers. There are STREAMS modules, which are pushable nondriver entities, and there are kernel-loadable modules, which are components of the kernel. See the appropriate chapters in "Writing Device Drivers for Oracle Solaris 11.2 ".

The kernel configuration mechanism must distinguish between STREAMS devices and traditional character devices because system calls to STREAMS drivers are processed by STREAMS routines, not by the system driver routines. The `streamtab` pointer in the `cb_ops`(9S) structure provides this distinction. If it is NULL, there are no STREAMS routines to execute; otherwise, STREAMS drivers initialize `streamtab` with a pointer to a `streamtab`(9S) structure containing the driver's STREAMS queue processing entry points.

STREAMS Initialization Entry Points

The initialization entry points of STREAMS drivers must perform the same tasks as those of non-STREAMS drivers. See "Writing Device Drivers for Oracle Solaris 11.2 " for more information.

STREAMS Table-Driven Entry Points

In non-STREAMS drivers, most of the driver's work is accomplished through the entry points in the `cb_ops`(9S) structure. For STREAMS drivers, most of the work is accomplished through the message-based STREAMS queue processing entry points.

Figure 9-1 shows multiple streams (corresponding to minor devices) connecting to a common driver. There are two distinct streams opened from the same major device. Consequently, they have the same streamtab and the same driver procedures.

FIGURE 9-1 Device Driver Streams
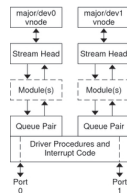
Multiple instances (minor devices) of the same driver are handled during the initial open for each device. Typically, a driver stores the queue address in a driver-private structure that is uniquely identified by the minor device number. (The DDI/DKI provides a mechanism for uniform handling of driver-private structures; see ddi_soft_state(9F)). The q_ptr of the queue points to the private data structure entry. When the messages are received by the queue, the calls to the driver put and service procedures pass the address of the queue, enabling the procedures to determine the associated device through the q_ptr field.

STREAMS guarantees that only one open or close can be active at a time per major/minor device pair.

STREAMS Queue Processing Entry Points

STREAMS device drivers have processing routines that are registered with the framework through the streamtab structure. The put procedure is a driver's entry point, but it is a message (not system) interface. STREAMS drivers and STREAMS modules implement these entry points similarly, as described in "Entry Points" on page 96.

The stream head translates write(2) and ioctl(2) calls into messages and sends them downstream to be processed by the driver's write queue put(9E) procedure. read is seen directly only by the stream head, which contains the functions required to process system calls. A STREAMS driver does not check system interfaces other than open and close, but it can detect the absence of a read indirectly if flow control propagates from the stream head to the driver and affects the driver's ability to send messages upstream.

For read-side processing, when the driver is ready to send data or other information to a user process, it prepares a message and sends it upstream to the read queue of the appropriate (minor

device) stream. The driver's open routine generally stores the queue address corresponding to this stream.

For write-side (or output) processing, the driver receives messages in place of a write call. If the message cannot be sent immediately to the hardware, it may be stored on the driver's write message queue. Subsequent output interrupts can remove messages from this queue.

A driver is at the end of a stream. As a result, drivers must include standard processing for certain message types that a module might be able to pass to the next component. For example, a driver must process all `M_IOCTL` messages; otherwise, the stream head blocks for an `M_IOCNAK`, `M_IOCACK`, or until the timeout (potentially infinite) expires. If a driver does not understand an `ioctl`(2), an `M_IOCNAK` message is sent upstream.

Messages that are not understood by the drivers should be freed.

The stream head locks up the stream when it receives an `M_ERROR` message, so driver developers should be careful when using the `M_ERROR` message.

STREAMS Interrupt Handlers

Most hardware drivers have an interrupt handler routine. You must supply an interrupt routine for the device's driver. The interrupt handling for STREAMS drivers is not fundamentally different from that for other device drivers. Drivers usually register interrupt handlers in their `attach`(9E) entry point, using `ddi_add_intr`(9F). Drivers unregister the interrupt handler at detach time using `ddi_remove_intr`(9F).

The system also supports software interrupts. The routines `ddi_add_softintr`(9F) and `ddi_remove_softintr`(9F) register and unregister (respectively) soft-interrupt handlers. A software interrupt is generated by calling `ddi_trigger_softintr`(9F).

See "Writing Device Drivers for Oracle Solaris 11.2 " for more information.

Driver Unloading

STREAMS drivers can prevent unloading through the standard driver `detach`(9E) entry point.

STREAMS Driver Code Samples

The following discussion describes characteristics of a STREAMS driver:

■   Basic hardware/pseudo drivers

This type of driver communicates with a specific piece of hardware (or simulated hardware). The `lp` example simulates a simple printer driver.

- Clonable drivers

  The STREAMS framework supports a `CLONEOPEN` facility, which allows multiple streams to be opened from a single special file. If a STREAMS device driver chooses to support `CLONEOPEN`, it can be referred to as a clonable device. The `attach(9E)` routines from two Oracle Solaris drivers, `ptm(7D)` and `log(7D)`, illustrate two approaches to cloning.

- Multiple instances in drivers

  A multiplexer driver is a regular STREAMS driver that can handle multiple streams connected to it instead of just one stream. Multiple connections occur when more than one minor device of the same driver is in use. See "Cloning STREAMS Drivers" on page 180 for more information.

Printer Driver Example

Example 9-1 is a sample print driver for an interrupt-per-character line printer. The driver is unidirectional—it has no read-side processing. It demonstrates some differences between module and driver programming, including the following:

- Declarations for driver configuration
- Open handling
- A driver is passed a device number
- Flush handling
- A driver must loop `M_FLUSH` messages back upstream
- Interrupt routine
- A driver registers interrupt handler and processes interrupts

Most of the STREAMS processing in the driver is independent of the actual printer hardware; in this example, actual interaction with the printer is limited to the `lpoutchar` function, which prints one character at a time. For purposes of demonstration, the "printer hardware" is actually the system console, accessed through `cmn_err(9F)`. Since there's no actual hardware to generate a genuine hardware interrupt, `lpoutchar` simulates interrupts using `ddi_trigger_softintr(9F)`. For a real printer, the `lpoutchar` function is rewritten to send a character to the printer, which should generate a hardware interrupt.

The driver declarations follow. After specifying header files (include <sys/ddi.h> and <sys/sunddi.h> as the last two header files), the driver declares a per-printer structure, `struct lp`. This structure contains members that enable the driver to keep track of each instance of the driver, such as `flags` (what the driver is doing), `msg` (the current STREAMS print message), `qptr` (pointer to the stream's write queue), `dip` (the instance's device information handle), `iblock_cookie` (for registering an interrupt handler), `siid` (the handle of the soft interrupt), and

lp_lock (a mutex to protect the data structure from multithreaded race conditions). The driver next defines the bits for the flags member of struct lp; the driver defines only one flag, BUSY.

Following function prototypes, the driver provides some standard STREAMS declarations: a module_info(9S) structure (minfo), a qinit(9S) structure for the read side (rinit) that is initialized by the driver's open and close entry points, a qinit(9S) structure for the write side (winit) that is initialized by the write put procedure, and a streamtab(9S) that points to rinit and winit. The values in the module name and ID fields in the module_info(9S) structure must be unique in the system. Because the driver is unidirectional, there is no read side put or service procedure. The flow control limits for use on the write side are 50 bytes for the low-watermark and 150 bytes for the high-watermark.

The driver next declares lp_state. This is an anchor on which the various "soft-state" functions provided by the DDK operate. The ddi_soft_state(9F) manual page describes how to maintain multiple instances of a driver.

The driver next declares a cb_ops(9S) structure, which is required in all device drivers. In non-STREAMS device drivers, cb_ops(9S) contains vectors to the table-driven entry points. For STREAMS drivers, however, cb_ops(9S) contains mostly nodev entries. The cb_stream field, however, is initialized with a pointer to the driver's streamtab(9S) structure. This indicates to the kernel that this driver is a STREAMS driver.

Next, the driver declares a dev_ops(9S) structure, which points to the various initialization entry points as well as to the cb_ops(9S) structure. Finally, the driver declares a struct moldrv and a struct modlinkage for use by the kernel linker when the driver is dynamically loaded. struct moldrv contains a pointer to mod_driverops (a significant difference between a STREAMS driver and a STREAMS module—a STREAMS module would contain a pointer to mod_strops instead).

EXAMPLE 9-1   Simple Line Printer Driver

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/signal.h>
#include <sys/errno.h>
#include <sys/cred.h>
#include <sys/stat.h>
#include <sys/modctl.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

/* This is a private data structure, one per minor device number */
```

```
struct lp {
 short flags; /* flags -- see below */
 mblk_t *msg; /* current message being output */
 queue_t *qptr; /* back pointer to write queue */
 dev_info_t *dip; /* devinfo handle */
 ddi_iblock_cookie_t iblock_cookie;
 ddi_softintr_t siid;
 kmutex_t lp_lock; /* sync lock */
};

/* flags bits */

#define BUSY 1 /* dev is running, int is forthcoming */

/*
 * Function prototypes.
 */
static int lpattach(dev_info_t *, ddi_attach_cmd_t);
static int lpdetach(dev_info_t *, ddi_detach_cmd_t);
static int lpgetinfo(dev_info_t *, ddi_info_cmd_t, void *, void **);
static int lpidentify(dev_info_t *);
static uint lpintr(caddr_t lp);
static void lpout(struct lp *lp);
static void lpoutchar(struct lp *lp, char c);
static int lpopen(queue_t*, dev_t*, int, int, cred_t*);
static int lpclose(queue_t*, int, cred_t*);
static int lpwput(queue_t*, mblk_t*);

/* Standard Streams declarations */

static struct module_info minfo = {
 0xaabb,
 "lp",
 0,
 INFPSZ,
 150,
 50
};

static struct qinit rinit = {
 (int (*)()) NULL,
 (int (*)()) NULL,
 lpopen,
 lpclose,
 (int (*)()) NULL,
 &minfo,
 NULL
};

static struct qinit winit = {
 lpwput,
 (int (*)()) NULL,
 (int (*)()) NULL,
 (int (*)()) NULL,
 (int (*)()) NULL,
 &minfo,
 NULL
};
```

```
static struct streamtab lpstrinfo = { &rinit, &winit, NULL, NULL };

/*
 * An opaque handle where our lp lives
 */
static void *lp_state;

/* Module Loading/Unloading and Autoconfiguration declarations */

static struct cb_ops lp_cb_ops = {
 nodev, /* cb_open */
 nodev, /* cb_close */
 nodev, /* cb_strategy */
 nodev, /* cb_print */
 nodev, /* cb_dump */
 nodev, /* cb_read */
 nodev, /* cb_write */
 nodev, /* cb_ioctl */
 nodev, /* cb_devmap */
 nodev, /* cb_mmap */
 nodev, /* cb_segmap */
 nochpoll, /* cb_chpoll */
 ddi_prop_op, /* cb_prop_op */
 &lpstrinfo, /* cb_stream */
 D_MP | D_NEW, /* cb_flag */
};

static struct dev_ops lp_ops = {
 DEVO_REV, /* devo_rev */
 0, /* devo_refcnt */
 lpgetinfo, /* devo_getinfo */
 lpidentify, /* devo_identify */
 nulldev, /* devo_probe */
 lpattach, /* devo_attach */
 lpdetach, /* devo_detach */
 nodev, /* devo_reset */
 &lp_cb_ops, /* devo_cb_ops */
 (struct bus_ops *)NULL /* devo_bus_ops */
};

/*
 * Module linkage information for the kernel.
 */
static struct modldrv modldrv = {
 &mod_driverops,
 "Simple Sample Printer Streams Driver", /* Description */
 &lp_ops, /* driver ops */
};

static struct modlinkage modlinkage = {
 MODREV_1, &modldrv, NULL
};
```

Example 9-2 shows the required driver configuration entry points _init(9E), _fini(9E), and _info(9E). In addition to installing the driver using mod_install(9F),

the `_init` entry point also initializes the per-instance driver structure using
ddi_soft_state_init(9F). _fini(9E) performs the complementary calls to
mod_remove(9F) and ddi_soft_state_fini(9F) to unload the driver and release the
resources used by the soft-state routines.

EXAMPLE 9-2   Configuration Entry Points

```
int
_init(void)
{
 int e;

 if ((e = ddi_soft_state_init(&lp_state,
  sizeof (struct lp), 1)) != 0) {
  return (e);
 }

 if ((e = mod_install(&modlinkage)) != 0) {
  ddi_soft_state_fini(&lp_state);
 }

 return (e);
}

int
_fini(void)
{
 int e;

 if ((e = mod_remove(&modlinkage)) != 0) {
  return (e);
 }
 ddi_soft_state_fini(&lp_state);
 return (e);
}

int
_info(struct modinfo *modinfop)
{
 return (mod_info(&modlinkage, modinfop));
}
```

Example 9-3 shows the `lp` driver's implementation of the initialization entry points. In
`lpidentify`, the driver ensures that the name of the device being attached is "lp".

`lpattach` first uses ddi_soft_state_zalloc(9F) to allocate a per-instance structure
for the printer being attached. Next it creates a node in the device tree for the printer using
ddi_create_minor_node(9F); user programs use the node to access the device. `lpattach`
then registers the driver interrupt handler because the sample is driver pseudo-hardware, the
driver uses soft interrupts. A driver for a real printer would use ddi_add_intr(9F) instead
of ddi_add_softintr(9F). A driver for a real printer would also need to perform any other

required hardware initialization in lpattach. Finally, lpattach initializes the per-instance mutex.

In lpdetach, the driver undoes everything it did in lpattach.

lpgetinfo uses the soft-state structures to obtain the required information.

EXAMPLE 9-3   Initialization Entry Points

```
static int
lpidentify(dev_info_t *dip)
{
 if (strcmp(ddi_get_name(dip), "lp") == 0) {
  return (DDI_IDENTIFIED);
 } else
  return (DDI_NOT_IDENTIFIED);
}

static int
lpattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
 int instance;
 struct lp *lpp;

 switch (cmd) {

 case DDI_ATTACH:

  instance = ddi_get_instance(dip);

  if (ddi_soft_state_zalloc(lp_state, instance) != DDI_SUCCESS) {
   cmn_err(CE_CONT, "%s%d: can't allocate state\n",
    ddi_get_name(dip), instance);
   return (DDI_FAILURE);
  } else
   lpp = ddi_get_soft_state(lp_state, instance);

  if (ddi_create_minor_node(dip, "strlp", S_IFCHR,
   instance, NULL, 0) == DDI_FAILURE) {
   ddi_remove_minor_node(dip, NULL);
   goto attach_failed;

  }

  lpp->dip = dip;
  ddi_set_driver_private(dip, (caddr_t)lpp);

  /* add (soft) interrupt */


  if (ddi_add_softintr(dip, DDI_SOFTINT_LOW, &lpp->siid,
   &lpp->iblock_cookie, 0, lpintr, (caddr_t)lpp)
   != DDI_SUCCESS) {
   ddi_remove_minor_node(dip, NULL);
   goto attach_failed;
```

```
  }

  mutex_init(&lpp->lp_lock, "lp lock", MUTEX_DRIVER,
   (void *)lpp->iblock_cookie);


  ddi_report_dev(dip);
  return (DDI_SUCCESS);

 default:
  return (DDI_FAILURE);
 }

attach_failed:
 /*
  * Use our own detach routine to toss
  * away any stuff we allocated above.
  */
 (void) lpdetach(dip, DDI_DETACH);
 return (DDI_FAILURE);
}

static int
lpdetach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
 int instance;
 struct lp *lpp;

 switch (cmd) {

 case DDI_DETACH:
  /*
   * Undo what we did in lpattach, freeing resources
   * and removing things we installed. The system
   * framework guarantees we are not active with this devinfo
   * node in any other entry points at this time.
   */
  ddi_prop_remove_all(dip);
  instance = ddi_get_instance(dip);
  lpp = ddi_get_soft_state(lp_state, instance);
  ddi_remove_minor_node(dip, NULL);
  ddi_remove_softintr(lpp->siid);
  ddi_soft_state_free(lp_state, instance);
  return (DDI_SUCCESS);

 default:
  return (DDI_FAILURE);
 }
}

/*ARGSUSED*/
static int
lpgetinfo(dev_info_t *dip, ddi_info_cmd_t infocmd, void *arg,
         void **result)
{
 struct lp *lpp;
 int error = DDI_FAILURE;
```

```
switch (infocmd) {
case DDI_INFO_DEVT2DEVINFO:
 if ((lpp = ddi_get_soft_state(lp_state,
  getminor((dev_t)arg))) != NULL) {
  *result = lpp->dip;
  error = DDI_SUCCESS;
 } else
  *result = NULL;
 break;

case DDI_INFO_DEVT2INSTANCE:
 *result = (void *)getminor((dev_t)arg);
 error = DDI_SUCCESS;
 break;

default:
 break;
}

 return (error);
}
```

The STREAMS mechanism allows only one stream per minor device. The driver open routine
is called whenever a STREAMS device is opened. open matches the correct private data
structure with the stream using ddi_get_soft_state(9F). The driver open, lpopen in
Example 9-4 , has the same interface as the module open.

The stream flag, sflag, must have the value 0, indicating a normal driver open. devp pointers to
the major/minor device number for the port. After checking sflag, lpopen uses devp to find the
correct soft-state structure.

The next check, if (q->q_ptr)..., determines if the printer is already open. q_ptr is a driver
or module private data pointer. It can be used by the driver for any purpose and is initialized to
zero by STREAMS before the first open. In this example, the driver sets the value of q_ptr, in
both the read and write queue structures, to point to the device's per-instance data structure. If
the pointer is non-NULL, it means the printer is already open, so lpopen returns EBUSY to avoid
merging printouts from multiple users.

The driver close routine is called by the stream head. Any messages left in the queue are
automatically removed by STREAMS. The stream is dismantled and data structures are
released.

EXAMPLE 9-4   Queue Processing Entry Points

```
/*ARGSUSED*/
static int
lpopen(
 queue_t *q, /* read queue */
 dev_t *devp,
 int flag,
 int sflag,
```

```
 cred_t *credp)

{

 struct lp *lp;

 if (sflag) /* driver refuses to do module or clone open */
   return (ENXIO);

 if ((lp = ddi_get_soft_state(lp_state, getminor(*devp))) == NULL)
   return (ENXIO);

 /* Check if open already. Can't have multiple opens */

 if (q->q_ptr) {
   return (EBUSY);
 }

 lp->qptr = WR(q);
 q->q_ptr = (char *) lp;
 WR(q)->q_ptr = (char *) lp;
 qprocson(q);
 return (0);
}

/*ARGSUSED*/
static int
lpclose(
 queue_t *q,  /* read queue */
 int flag,
 cred_t *credp)
{

 struct lp *lp;

 qprocsoff(q);
 lp = (struct lp *) q->q_ptr;

 /*
  * Free message, queue is automatically
  * flushed by STREAMS
  */
 mutex_enter(&lp->lp_lock);

 if (lp->msg) {
   freemsg(lp->msg);
   lp->msg = NULL;
 }

 lp->flags = 0;
 mutex_exit(&lp->lp_lock);

 return (0);
}
```

There are no physical pointers between the read and write queue of a pair. WR(9F) is a queue pointer function. WR(9F) generates the write pointer from the read pointer. RD(9F) and

OTHERQ(9F) are additional queue pointer functions. RD(9F) generates the read pointer from the write pointer, and OTHERQ(9F) generates the mate pointer from either.

Driver Flush Handling

The write put procedure in Example 9-5 , lpwput, illustrates driver M_FLUSH handling. Note that all drivers are expected to incorporate flush handling.

If FLUSHW is set, the write message queue is flushed, and (in this example) the leading message (lp->msg) is also flushed. lp_lock protects the driver's per-instance data structure.

In most drivers, if FLUSHR is set, the read queue is flushed. However, in this example, no messages are ever placed on the read queue, so flushing it is not necessary. The FLUSHW bit is cleared and the message is sent upstream using qreply(9F). If FLUSHR is not set, the message is discarded.

The stream head always performs the following actions on flush requests received on the read side from downstream. If FLUSHR is set, messages waiting to be sent to user space are flushed. If FLUSHW is set, the stream head clears the FLUSHR bit and sends the M_FLUSH message downstream. In this manner, a single M_FLUSH message sent from the driver can reach all queues in a stream. A module must send two M_FLUSH messages to have the same effect.

lpwput queues M_DATA and M_IOCTL messages and if the device is not busy, starts output by calling lpout. Message types that are not recognized are discarded (in the default case of the switch).

EXAMPLE 9-5 Driver Flush Handling

```
static int lpwput(
 queue_t *q, /* write queue */
 mblk_t *mp) /* message pointer */
{
 struct lp *lp;

 lp = (struct lp *)q->q_ptr;

 switch (mp->b_datap->db_type) {
 default:
  freemsg(mp);
  break;

 case M_FLUSH: /* Canonical flush handling */
  if (*mp->b_rptr & FLUSHW) {
   flushq(q, FLUSHDATA);
   mutex_enter(&lp->lp_lock); /* lock any access to lp */

   if (lp->msg) {
    freemsg(lp->msg);
```

```
  lp->msg = NULL;
 }

 mutex_exit(&lp->lp_lock);

}

if (*mp->b_rptr & FLUSHR) {
 *mp->b_rptr &= ~FLUSHW;
 qreply(q, mp);
} else
 freemsg(mp);

break;

case M_IOCTL:
case M_DATA:
 (void) putq(q, mp);
 mutex_enter(&lp->lp_lock);

 if (!(lp->flags & BUSY))
  lpout(lp);

 mutex_exit(&lp->lp_lock);

}
return (0);
}
```

Print Driver Interrupt

Example 9-6 shows the interrupt handling for the printer driver.

lpintr is the driver-interrupt handler registered by the attach routine.

lpout takes a single character from the queue and sends it to the printer. For convenience, the message currently being output is stored in lp->msg in the per-instance structure. This assumes that the message is called with the mutex held.

lpoutchar sends a single character to the printer (in this case the system console using cmn_err(9F)) and interrupts when complete. Of course, hardware would generate a hard interrupt, so the call to ddi_trigger_softintr(9F) would be unnecessary.

EXAMPLE 9-6   Driver Interrupt Handling

```
/* Device interrupt routine */static uint
lpintr(caddr_t lp)  /* minor device number of lp */
{

 struct lp *lpp = (struct lp *)lp;
```

```
  mutex_enter(&lpp->lp_lock);

  if (!(lpp->flags & BUSY)) {
    mutex_exit(&lpp->lp_lock);
    return (DDI_INTR_UNCLAIMED);
  }

  lpp->flags &= ~BUSY;
  lpout(lpp);
  mutex_exit(&lpp->lp_lock);

  return (DDI_INTR_CLAIMED);
}

/* Start output to device - used by put procedure and driver */

static void
lpout(
 struct lp *lp)
{

 mblk_t *bp;
 queue_t *q;

 q = lp->qptr;

 loop:
 if ((bp = lp->msg) == NULL) { /*no current message*/
   if ((bp = getq(q)) == NULL) {
     lp->flags &= ~BUSY;
     return;
   }
   if (bp->b_datap->db_type == M_IOCTL) {
    /* lpdoioctl(lp, bp); */
     goto loop;
   }

   lp->msg = bp; /* new message */

  }

 if (bp->b_rptr >= bp->b_wptr) { /* validate message */

   bp = lp->msg->b_cont;
   lp->msg->b_cont = NULL;
   freeb(lp->msg);
   lp->msg = bp;
   goto loop;
 }

 lpoutchar(lp, *bp->b_rptr++); /*output one character*/
 lp->flags |= BUSY;
}

static void
lpoutchar(
 struct lp *lp,
 char c)
```

```
{
 cmn_err(CE_CONT, "%c", c);
 ddi_trigger_softintr(lp->siid);
}
```

Driver Flow Control

The same utilities (described in Chapter 10, "STREAMS Modules") and mechanisms used for module flow control are used by drivers.

When the message is queued, `putq`(9F) increments the value of `q_count` by the size of the message and compares the result to the driver's write high-watermark (`q_hiwat`) value. If the count reaches `q_hiwat`, `putq`(9F) sets the internal `FULL` indicator for the driver write queue. This causes messages from upstream to be halted (`canputnext`(9F) returns FALSE) until the write queue count drops below `q_lowat`. The driver messages waiting to be output through `lpout` are dequeued by the driver output interrupt routine with `getq`(9F), which decrements the count. If the resulting count is below `q_lowat`, `getq`(9F) back-enables any upstream queue that had been blocked.

For priority band data, `qb_count`, `qb_hiwat`, and `qb_lowat` are used.

STREAMS with flow control can be used on the driver read side to handle temporary upstream blocks.

To some extent, a driver or a module can control when its upstream transmission becomes blocked. Control is available through the `M_SETOPTS` message (see Appendix A, "Message Types") to modify the stream head read-side flow control limits.

Cloning STREAMS Drivers

To eliminate polling, STREAMS drivers can be made clonable. If a STREAMS driver is implemented as a clonable device, a single node in the file system can be opened to access any unused device that the driver controls. This special node guarantees that each user is allocated a separate stream to the driver for each `open` call. Each stream is associated with an unused minor device, so the total number of streams that may be connected to a particular clonable driver is limited only by the number of minor devices configured for that driver.

In previous examples, each user process connected a stream to a driver by explicitly opening a particular minor device of the driver. Each minor device had its own node in the device tree file system. Often, there is a need for a user process to connect a new stream to a driver regardless of which minor device is used to access the driver. In the past, this forced the user process to poll the various minor device nodes of the driver for an available minor device.

The clone model is useful, for example, in a networking environment where a protocol pseudo-device driver requires each user to open a separate stream over which it establishes

communication. (The decision to implement a STREAMS driver as a clonable device is made by the designers of the device driver. Knowledge of the clone driver implementation is not required to use it.)

There are two ways to open as a clone device. The first is to use the STREAMS framework-provided clone device, which arranges to open the device with the CLONEOPEN flag passed in. This method is demonstrated in Example 9-7 , which shows the attach and open routines for the pseudo-terminal master ptm(7D) driver. The second way is to have the driver open itself as a clone device, without intervention from the system clone device. This method is demonstrated in the attach and open routines for the log(7D) device in Example 9-8 .

The ptm(7D) device, which uses the system-provided clone device, sets up two nodes in the device file system. One has a major number of 23 (ptm's assigned major number) and a minor number of 0. The other node has a major number of 11 (the clone device's assigned major number) and a minor number of 23 (ptm's assigned major number). The driver's attach routine (see Example 9-7) calls to ddi_create_minor_node(9F) twice. First, to set up the "normal" node (major number 23); second, to specify CLONE_DEV as the last parameter, making the system create the node with major 11.

```
crw-rw-rw-   1 sys       11, 23 Mar  6 02:05 clone:ptmx
crw-------   1 sys       23,  0 Mar  6 02:05 ptm:ptmajor
```

When the special file /devices/pseudo/clone@0:ptmx is opened, the clone driver code in the kernel (accessed by major 11) passes the CLONEOPEN flag in the sflag parameter to the ptm(7D) open routine. ptm's open routine checks sflag to make sure it is being called by the clone driver. The open routine next attempts to find an unused minor device for the open by searching its table of minor devices. (PT_ENTER_WRITE and PT_EXIT_WRITE are driver-defined macros for entering and exiting the driver's mutex.) If it succeeds (and following other open processing), the open routine constructs a new dev_t with the new minor number, which it passes back to its caller in the devp parameter. (The new minor number is available to the user program that opened the clonable device through an fstat(2) call.)

EXAMPLE 9-7   Opening a System Clone Device

```
static int
ptm_attach(dev_info_t *devi, ddi_attach_cmd_t cmd)
{
   if (cmd != DDI_ATTACH)
   return (DDI_FAILURE);

   if (ddi_create_minor_node(devi, "ptmajor", S_IFCHR, 0, NULL, 0)
    == DDI_FAILURE) {
   ddi_remove_minor_node(devi, NULL);
   return (DDI_FAILURE);
   }
   if (ddi_create_minor_node(devi, "ptmx", S_IFCHR, 0, NULL, CLONE_DEV)
```

```
      == DDI_FAILURE) {
     ddi_remove_minor_node(devi, NULL);
     return (DDI_FAILURE);
     }
     ptm_dip = devi;
     return (DDI_SUCCESS);
}

static int
ptmopen(
  queue_t *rqp,  /* pointer to the read side queue */
  dev_t *devp,  /* pointer to stream tail's dev */
  int  oflag,  /* the user open(2) supplied flags */
  int  sflag,  /* open state flag */
  cred_t *credp) /* credentials */
{
  struct pt_ttys *ptmp;
  mblk_t *mop;  /* ptr to a setopts message block */
  minor_t dev;

  if (sflag != CLONEOPEN) {
   return (EINVAL);
  }

  for (dev = 0; dev < pt_cnt; dev++) {
   ptmp = &ptms_tty[dev];
   PT_ENTER_WRITE(ptmp);
   if (ptmp->pt_state & (PTMOPEN | PTSOPEN | PTLOCK)) {
    PT_EXIT_WRITE(ptmp);
   } else
    break;
  }

  if (dev >= pt_cnt) {
   return (ENODEV);
  }

  ... <other open processing> ...

  /*
   * The input, devp, is a major device number, the output is put into
   * into the same parm as a major,minor pair.
   */
  *devp = makedevice(getmajor(*devp), dev);
  return (0);
}
```

The log(7D) driver uses the second method; it clones itself without intervention from the system clone device. The log(7D) driver's attach routine (in Example 9-8) is similar to the one in ptm(7D). It creates two nodes using ddi_create_minor_node(9F), but neither specifies CLONE_DEV as the last parameter. Instead, one of the devices has minor 0, the other minor CLONEMIN. These two devices provide log(7D) two interfaces: the first write-only, the second read-write (see the man page log(7D) for more information). Users open one node or the other. If they open the CONSWMIN (clonable, read-write) node, the open routine checks

its table of minor devices for an unused device. If it is successful, it (like the `ptm(7D)` open routine) returns the new `dev_t` to its caller in `devp`.

EXAMPLE 9-8 Opening the `log` Driver

```
static int
log_attach(dev_info_t *devi, ddi_attach_cmd_t cmd)
{
  if (ddi_create_minor_node(devi, "conslog", S_IFCHR, 0, NULL, NULL)
      == DDI_FAILURE ||
      ddi_create_minor_node(devi, "log", S_IFCHR, 5, NULL, NULL)
      == DDI_FAILURE) {
   ddi_remove_minor_node(devi, NULL);
   return (-1);
  }
  log_dip = devi;
  return (DDI_SUCCESS);
}

static int
logopen(
  queue_t *q,
  dev_t *devp,
  int flag,
  int sflag,
  cred_t *cr)
{
  int i;
  struct log *lp;

  /*
   * A MODOPEN is invalid and so is a CLONEOPEN.
   * This is because a clone open comes in as a CLONEMIN device open!!
   */
  if (sflag)
   return (ENXIO);

  mutex_enter(&log_lock);
  switch (getminor(*devp)) {

  case CONSWMIN:
   if (flag & FREAD) { /* you can only write to this minor */
    mutex_exit(&log_lock);
    return (EINVAL);
   }
   if (q->q_ptr) { /* already open */
    mutex_exit(&log_lock);
    return (0);
   }
   lp = &log_log[CONSWMIN];
   break;

  case CLONEMIN:
   /*
    * Find an unused minor > CLONEMIN.
```

```
         */
        i = CLONEMIN + 1;
       for (lp = &log_log[i]; i < log_cnt; i++, lp++) {
        if (!(lp->log_state & LOGOPEN))
          break;
       }
       if (i >= log_cnt) {
        mutex_exit(&log_lock);
         return (ENXIO);
       }
       *devp = makedevice(getmajor(*devp), i); /* clone it */
       break;

     default:
      mutex_exit(&log_lock);
      return (ENXIO);
     }

     /*
      * Finish device initialization.
      */
     lp->log_state = LOGOPEN;
     lp->log_rdq = q;
     q->q_ptr = (void *)lp;
     WR(q)->q_ptr = (void *)lp;
     mutex_exit(&log_lock);
     qprocson(q);
     return (0);
}
```

Loop-Around Driver

The loop-around driver is a pseudo-driver that loops data from one open stream to another open stream. The associated files are almost like a full-duplex pipe to user processes. The streams are not physically linked. The driver is a simple multiplexer that passes messages from one stream's write queue to the other stream's read queue.

To create a connection, a process opens two streams, obtains the minor device number associated with one of the returned file descriptors, and sends the device number in an

ioctl(2) to the other stream. For each open, the driver open places the passed queue pointer in a driver interconnection table, indexed by the device number. When the driver later receives an M_IOCTL message, it uses the device number to locate the other stream's interconnection table entry, and stores the appropriate queue pointers in both of the streams' interconnection table entries.

Subsequently, when messages other than M_IOCTL or M_FLUSH are received by the driver on either stream's write side, the messages are switched to the read queue following the driver on the other stream's read side. The resultant logical connection is shown in Figure 9-2. Flow control between the two streams must be handled explicitly, since STREAMS do not automatically propagate flow control information between two streams that are not physically connected.

FIGURE 9-2   Loop-Around Streams
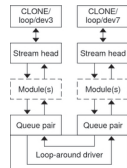
Example 9-9 shows the loop-around driver code. The `loop` structure contains the interconnection information for a pair of streams. `loop_loop` is indexed by the minor device number. When a stream is opened to the driver, the driver places the address of the corresponding `loop_loop` element in the `q_ptr` (private data structure pointer) of the read-side and write-side queues. Since STREAMS clears `q_ptr` when the queue is allocated, a NULL value of `q_ptr` indicates an initial open. `loop_loop` verifies that this stream is connected to another `open` stream.

The code presented here for the loop-around driver represents a single-threaded, uniprocessor implementation. Chapter 12, "Multithreaded STREAMS" presents multiprocessor and multithreading issues such as locking to prevent race conditions and data corruption.

Example 9-9 contains the declarations for the driver.

EXAMPLE 9-9   Declarations for the Loop-Around Driver

```
/* Loop-around driver */

#include <sys/types.h>
#include <sys/param.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/signal.h>
#include <sys/errno.h>
#include <sys/cred.h>
#include <sys/stat.h>
#include <sys/modctl.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

static int loop_identify(dev_info_t *);
static int loop_attach(dev_info_t *, ddi_attach_cmd_t);
static int loop_detach(dev_info_t *, ddi_detach_cmd_t);
static int loop_devinfo(dev_info_t *, ddi_info_cmd_t, void *, void **);
static int loopopen (queue_t*, dev_t*, int, int, cred_t*);
static int loopclose (queue_t*, int, cred_t*);
static int loopwput (queue_t*, mblk_t*);
static int loopwsrv (queue_t*);
```

```
static int looprsrv (queue_t*);

static dev_info_t *loop_dip; /* private devinfo pointer */

static struct module_info minfo = {
  0xee12,
  "loop",
  0,
  INFPSZ,
  512,
  128
};

static struct qinit rinit = {
  (int (*)()) NULL,
  looprsrv,
  loopopen,
  loopclose,
  (int (*)()) NULL,
  &minfo,
  NULL
};

static struct qinit winit = {
  loopwput,
  loopwsrv,
  (int (*)()) NULL,
  (int (*)()) NULL,
  (int (*)()) NULL,
  &minfo,
  NULL
};

static struct streamtab loopinfo= {
  &rinit,
  &winit,
  NULL,
  NULL
};

struct  loop {
  queue_t *qptr;   /* back pointer to write queue */
  queue_t *oqptr;  /* pointer to connected read queue */
};

#define LOOP_CONF_FLAG (D_NEW | D_MP)

static struct cb_ops cb_loop_ops = {
    nulldev,              /* cb_open */
    nulldev,              /* cb_close */
    nodev,                /* cb_strategy */
    nodev,                /* cb_print */
    nodev,                /* cb_dump */
    nodev,                /* cb_read */
    nodev,                /* cb_write */
    nodev,                /* cb_ioctl */
    nodev,                /* cb_devmap */
    nodev,                /* cb_mmap */
```

```
   nodev,                  /* cb_segmap */
   nochpoll,               /* cb_chpoll */
   ddi_prop_op,            /* cb_prop_op */
   ( &loopinfo),           /* cb_stream */
   (int)(LOOP_CONF_FLAG)   /* cb_flag */
};

static struct dev_ops loop_ops = {
   DEVO_REV,               /* devo_rev */
   0,                      /* devo_refcnt */
   (loop_devinfo),         /* devo_getinfo */
   (loop_identify),        /* devo_identify */
   (nulldev),              /* devo_probe */
   (loop_attach),          /* devo_attach */
   (loop_detach),          /* devo_detach */
   (nodev),                /* devo_reset */
   &(cb_loop_ops),         /* devo_cb_ops */
   (struct bus_ops *)NULL, /* devo_bus_ops */
   (int (*)()) NULL        /* devo_power */
};

#define LOOP_SET ((`l'<<8)|1) /* in a .h file */
#define NLOOP 64
static struct loop loop_loop[NLOOP];
static int loop_cnt = NLOOP;

/*
 * Module linkage information for the kernel.
 */
extern struct mod_ops mod_strmodops;

static struct modldrv modldrv = {
  &mod_driverops, "STREAMS loop driver", &loop_ops
};

static struct modlinkage modlinkage = {
  MODREV_1, &modldrv, NULL
};

_init()
{
  return (mod_install(&modlinkage));
}

_info(modinfop)
  struct modinfo *modinfop;
{
  return (mod_info(&modlinkage, modinfop));
}

_fini(void)
{
  return (mod_remove(&modlinkage));
}
```

Example 9-10 contains the initialization routines.

EXAMPLE 9-10 Initialization Routines for the Loop-around Driver

```
static int
loop_identify(dev_info_t *devi)
{
   if (strcmp(ddi_get_name(devi), "loop") == 0)
   return (DDI_IDENTIFIED);
   else
   return (DDI_NOT_IDENTIFIED);
}

static int
loop_attach(dev_info_t *devi, ddi_attach_cmd_t cmd)
{
   if (cmd != DDI_ATTACH)
   return (DDI_FAILURE);

   if (ddi_create_minor_node(devi, "loopmajor", S_IFCHR, 0, NULL, 0)
    == DDI_FAILURE) {
   ddi_remove_minor_node(devi, NULL);
   return (DDI_FAILURE);
   }
   if (ddi_create_minor_node(devi, "loopx", S_IFCHR, 0, NULL, CLONE_DEV)
     == DDI_FAILURE) {
   ddi_remove_minor_node(devi, NULL);
   return (DDI_FAILURE);
   }

   loop_dip = devi;

   return (DDI_SUCCESS);
}

static int
loop_detach(dev_info_t *devi, ddi_detach_cmd_t cmd)
{
   if (cmd != DDI_DETACH)
   return (DDI_FAILURE);

   ddi_remove_minor_node(devi, NULL);
   return (DDI_SUCCESS);
}

/*ARGSUSED*/
static int
loop_devinfo(
   dev_info_t *dip,
   ddi_info_cmd_t infocmd,
   void *arg,
   void **result)
{
   int error;

   switch (infocmd) {
   case DDI_INFO_DEVT2DEVINFO:
   if (loop_dip == NULL) {
    error = DDI_FAILURE;
```

```
  } else {
   *result = (void *) loop_dip;
   error = DDI_SUCCESS;
  }
  break;
  case DDI_INFO_DEVT2INSTANCE:
  *result = (void *)0;
  error = DDI_SUCCESS;
  break;
  default:
  error = DDI_FAILURE;
  }
  return (error);

}
```

The open procedure (in Example 9-11) includes canonical clone processing that enables a single file system node to yield a new minor device/vnode each time the driver is opened. In `loopopen`, `sflag` can be `CLONEOPEN`, indicating that the driver picks an unused minor device. In this case, the driver scans its private `loop_loop` data structure to find an unused minor device number. If `sflag` is not set to `CLONEOPEN`, the passed-in minor device specified by `getminor(*devp)` is used.

EXAMPLE 9-11 Opening the Loop-Around Driver

```
/*ARGSUSED*/
static int loopopen(
  queue_t *q,
  dev_t *devp,
  int flag,
  int sflag,
  cred_t *credp)
{
  struct loop *loop;
  minor_t newminor;

  if (q->q_ptr) /* already open */
   return(0);

  /*
   * If CLONEOPEN, pick a minor device number to use.
   * Otherwise, check the minor device range.
   */

  if (sflag == CLONEOPEN) {
   for (newminor = 0; newminor < loop_cnt; newminor++ ) {
    if (loop_loop[newminor].qptr == NULL)
     break;
   }
  } else
   newminor = getminor(*devp);

  if (newminor >= loop_cnt)
   return(ENXIO);
```

```
    /*
     * construct new device number and reset devp
     * getmajor gets the major number
     */

    *devp = makedevice(getmajor(*devp), newminor);
    loop = &loop_loop[newminor];
    WR(q)->q_ptr = (char *) loop;
    q->q_ptr = (char *) loop;
    loop->qptr = WR(q);
    loop->oqptr = NULL;

    qprocson(q);

    return(0);
}
```

Because the messages are switched to the read queue following the other stream's read side, the driver needs a put procedure only on its write side. loopwput (in Example 9-12) shows another use of an ioctl(2). The driver supports the ioc_cmd value LOOP_SET in the iocblk(9S) of the M_IOCTL message. LOOP_SET makes the driver connect the current open stream to the stream indicated in the message. The second block of the M_IOCTL message holds an integer that specifies the minor device number of the stream to which to connect.

The LOOP_SET ioctl(2) processing involves several checks:

- Does the second block have the proper amount of data?
- Is the "to" device in range?
- Is the "to" device open?
- Is the current stream disconnected?
- Is the "to" stream disconnected?

If these checks pass, the read queue pointers for the two streams are stored in the respective oqptr fields. This cross-connects the two streams indirectly, through loop_loop.

The put procedure incorporates canonical flush handling.

loopwput queues all other messages (for example, M_DATA or M_PROTO) for processing by its service procedure. A check is made that the stream is connected. If not, M_ERROR is sent to the stream head. Certain message types can be sent upstream by drivers and modules to the stream head where they are translated into actions detectable by user processes. These messages may also modify the state of the stream head:

M_ERROR   Causes the stream head to lock up. Message transmission between stream and user processes is terminated. All subsequent system calls except close(2) and poll(2) fail. Also causes M_FLUSH, clearing all message queues, to be sent downstream by the stream head.

M_HANGUP   Terminates input from a user process to the stream. All subsequent
                        system calls that would send messages downstream fail. Once the stream
                        head read message queue is empty, EOF is returned on reads. This can
                        also result in SIGHUP being sent to the process group's session leader.

M_SIG/M_PCSIG   Causes a specified signal to be sent to the process group associated with
                        the stream.

putnextctl(9F) and putnextctl1(9F) allocate a nondata (that is, not M_DATA, M_DELAY,
M_PROTO, or M_PCPROTO) type message, place one byte in the message (for putnextctl1(9F)),
and call the put(9E) procedure of the specified queue.

EXAMPLE 9-12 Use of ioctl to Copy Data From User Space to Kernel Space

```
static int loopwput(queue_t *q, mblk_t *mp)
{
  struct loop *loop;
  int to;

  loop = (struct loop *)q->q_ptr;

  switch (mp->b_datap->db_type) {
   case M_IOCTL: {
    struct iocblk *iocp;
    int   error=0;

    iocp = (struct iocblk *)mp->b_rptr;

    switch (iocp->ioc_cmd) {

      case LOOP_SET: {
      /*
       * if this is a transparent ioctl return an error;
       * the complete solution is to convert the message
       * into an M_COPYIN message so that the data is
       * ultimately copied from user space
       * to kernel space.
       */

      if (iocp->ioc_count == TRANSPARENT) {
       error = EINVAL;
       goto iocnak;
      }

      /* fetch other minor device number */

      to = *(int *)mp->b_cont->b_rptr;

      /*
       * Sanity check. ioc_count contains the amount
       * of user supplied data which must equal the
       * size of an int.
```

```
 */

if (iocp->ioc_count != sizeof(int)) {
 error = EINVAL;
 goto iocnak;
}

/* Is the minor device number in range? */

if (to >= loop_cnt || to < 0) {
 error = ENXIO;
 goto iocnak;
}

/* Is the other device open? */

if (!loop_loop[to].qptr) {
 error = ENXIO;
 goto iocnak;
}

/* Check if either dev is currently connected */

if (loop->oqptr || loop_loop[to].oqptr) {
 error = EBUSY;
 goto iocnak;
}

/* Cross connect the streams through
 * the loopstruct
 */

loop->oqptr = RD(loop_loop[to].qptr);
loop_loop[to].oqptr = RD(q);

/*
 * Return successful ioctl. Set ioc_count
 * to zero, since no data is returned.
 */

mp->b_datap->db_type = M_IOCACK;
iocp->ioc_count = 0;
qreply(q, mp);
break;
}

default:
 error = EINVAL;
 iocnak:

/*
 * Bad ioctl. Setting ioc_error causes
 * the ioctl call to return that particular errno.
 * By default, ioctl returns EINVAL on failure.
 */

mp->b_datap->db_type = M_IOCNAK;
iocp->ioc_error = error;
```

```
      qreply(q, mp);
      break;
     }

     break;
    }

    case M_FLUSH: {

     if (*mp->b_rptr & FLUSHW) {
      flushq(q, FLUSHALL); /* write */
      if (loop->oqptr)
       flushq(loop->oqptr, FLUSHALL);
      /* read on other side equals write on this side */
     }

     if (*mp->b_rptr & FLUSHR) {
      flushq(RD(q), FLUSHALL);
      if (loop->oqptr != NULL)
       flushq(WR(loop->oqptr), FLUSHALL);
     }

     switch(*mp->b_rptr) {

     case FLUSHW:
      *mp->b_rptr = FLUSHR;
      break;

     case FLUSHR:
      *mp->b_rptr = FLUSHW;
      break;

     }

     if (loop->oqptr != NULL)
      (void) putnext(loop->oqptr, mp);
     break;
    }

    default: /* If this Stream isn't connected, send
         * M_ERROR upstream.
         */
     if (loop->oqptr == NULL) {
      freemsg(mp);
      (void) putnextctl1(RD(q), M_ERROR, ENXIO);
      break;
     }
     (void) putq(q, mp);

    }

  return (0);
}
```

Service procedures are required in this example on both the write side and read side for flow control (see Example 9-13). The write service procedure, loopwsrv, takes on the canonical

form. The queue being written to is not downstream, but upstream (found through `oqptr`) on the other stream.

In this case, there is no read-side `put` procedure so the read `service` procedure, `looprsrv`, is not scheduled by an associated `put` procedure, as has been done previously. `looprsrv` is scheduled only by being back-enabled when its upstream flow control blockage is released. The purpose of the procedure is to re-enable the writer (`loopwsrv`) by using `oqptr` to find the related queue. `loopwsrv` cannot be directly back-enabled by STREAMS because there is no direct queue linkage between the two streams. Note that no message is queued to the read `service` procedure. Messages are kept on the write side so that flow control can propagate up to the stream head. `qenable`(9F) schedules the write-side `service` procedure of the other stream.

EXAMPLE 9-13 Loop-Around Driver Flow Control

```
staticintloopwsrv (queue_t*q)
{
  mblk_t *mp;
  structloop *loop;
  loop = (structloop*) q->q_ptr;

  while ((mp = getq (q)) != NULL){
    /* Check if we can put the message up
     * the other Stream read queue
     */
    if (mp->b_datap->db_type <= QPCTL && !canputnext (loop->oqptr)) {
    (void) putbq (q,mp);    /*read-side is blocked*/
    break;
    }

    /* send message to queue following other Stream read queue */

    (void) putnext (loop->oqptr, mp);
  }
  return (0);
}

staticintlooprsrv (queue_t*q)
{
  /* Enter only when "backenabled" by flow control */
  structloop *loop;
  loop = (structloop*) q->q_ptr;
  if (loop->oqptr == NULL)
   return (0);

  /*manually enable write service procedure*/
  qenable (WR (loop->oqptr));
  return (0);
}
```

`loopclose` breaks the connection between the streams, as shown in Example 9-14. `loopclose` sends an `M_HANGUP` message up the connected stream to the stream head.

EXAMPLE 9-14 Breaking Stream Connections for Loop-Around Device

```
/*ARGSUSED*/
static int loopclose (
  queue_t *q,
  int flag,
  cred_t *credp)
{
  struct loop *loop;

  loop = (struct loop *)q->q_ptr;
  loop->qptr = NULL;

  /*
   * If we are connected to another stream, break the linkage, and
   * send a hangup message. The hangup message causes the stream head
   * to reject writes, allow the queued data to be read completely,
   * and then return EOF on subsequent reads.
   */

  if (loop->oqptr) {
   (void) putnextctl(loop->oqptr, M_HANGUP);
   ((struct loop *)loop->oqptr->q_ptr)->oqptr = NULL;
   loop->oqptr = NULL;
  }

  qprocsoff(q);
  return (0);
}
```

An application using this driver would first open the clone device node created in the attach routine (/devices/pseudo/clone@0:loopx) two times to obtain two streams. The application can determine the minor numbers of the devices by using fstat(2). Next, it joins the two streams by using the STREAMS I_STR ioctl(2) (see streamio(7I)) to pass the LOOP_SET ioctl(2) with one of the stream's minor numbers as an argument to the other stream. Once this is completed, the data sent to one stream using write(2) or putmsg(2) can be retrieved from the other stream with read(2) or getmsg(2). The application also can interpose STREAMS modules between the stream heads and the driver using the I_PUSH ioctl(2).

Summarizing STREAMS Device Drivers

STREAMS device drivers are in many ways similar to non-STREAMS device drivers. The following points summarize the differences between STREAMS drivers and other drivers:

- Drivers must have attach(9E) and probe(9E) entry points to initialize the driver. The attach routine initializes the driver. Software drivers usually have little to initialize, because there is no hardware involved.

- Drivers have open(9E) and close(9E) routines.

- Most drivers have an interrupt handler routine. The driver developer is responsible for supplying an interrupt routine for the device's driver. In addition to hardware interrupts, the system also supports software interrupts. A software interrupt is generated by calling `ddi_trigger_softintr`(9F).

- All minor nodes are generated by `ddi_create_minor_node`(9F).

STREAMS device drivers also are similar to STREAMS modules. The following points summarize some of the differences between STREAMS modules and drivers.

- Messages that are not understood by the drivers should be freed.

- A driver must process all `M_IOCTL` messages. Otherwise, the stream head blocks for an `M_IOCNAK`, `M_IOCACK`, or until the timeout (potentially infinite) expires.

- If a driver does not understand an `ioctl`(2), an `M_IOCNAK` message must be sent upstream.

- The stream head locks up the stream when it receives an `M_ERROR` message, so driver developers should be careful when using the `M_ERROR` message.

- A hardware driver must provide an interrupt routine.

- Multithreaded drivers must protect their own data structures.

For more information on global driver issues and non-STREAMS drivers, see "Writing Device Drivers for Oracle Solaris 11.2 ".

10

STREAMS Modules

This chapter provides specific examples of how modules work, including code samples.

Module Overview

STREAMS modules process messages as they flow through the stream between an application and a character device driver. A STREAMS module is a pair of initialized queue structures and the specified kernel-level procedures that process data, status, and control information for the two queues. A stream can contain zero or more modules. Application processes push (stack) modules on a stream using the I_PUSH ioctl(2) and pop (unstack) them using the I_POP ioctl(2).

STREAMS Module Configuration

Like device drivers, STREAMS modules are dynamically linked and can be loaded into and unloaded from the running kernel.

Note - The word *module* is used differently when talking about drivers. A device driver is a kernel-loadable module that provides the interface between a device and the Device Driver Interface, and is linked to the kernel when it is first invoked.

A loadable module must provide linkage information to the kernel in an initialized modlstrmod(9S) and three entry points: _init(9E), _info(9E), and _fini(9E).

STREAMS modules can be unloaded from the kernel when not pushed onto a stream. A STREAMS module can prevent itself from being unloaded by returning an error (selected from errno.h) from its _fini(9E) routine (EBUSY is a good choice).

Module Procedures

STREAMS module procedures (`open`, `close`, `put`, `service`) have already been described in the previous chapters. This section shows some examples and further describes attributes common to module `put` and `service` procedures.

A module's `put` procedure is called by the preceding module, driver, or stream head, and always before that queue's `service` procedure. The `put` procedure does any immediate processing (for example, high-priority messages), while the corresponding `service` procedure performs deferred processing.

The `service` procedure is used primarily for performing deferred processing, with a secondary task to implement flow control. Once the `service` procedure is enabled, it can start but not finish before running user-level code. The `put` and `service` procedures must not block because there is no thread synchronization being done.

Example 10-1 shows a STREAMS module read-side `put` procedure.

EXAMPLE 10-1   Read-side `put` Procedure

```
static int
modrput (queue_t *q, mblk_t *mp)
{
  struct mod_prv *modptr;

  modptr = (struct mod_prv *) q->q_ptr;  /*state info*/

  if (mp->b_datap->db_type >= QPCTL){ /*proc pri msg*/
   putnext(q, mp);        /* and pass it on */
   return (0);
  }

  switch(mp->b_datap->db_type) {
   case M_DATA:    /* can process message data */
    putq(q, mp); /* queue msg for service procedure */
   return (0);

   case M_PROTO:   /* handle protocol control message */
     .
     .
     .

   default:
     putnext(q, mp);
     return (0);
  }
}
```

The preceding code does the following:

- A pointer to a queue defining an instance of the module and a pointer to a message are passed to the put procedure.

- The put procedure performs a switch on the type of the message. For each message type, the put procedure either enqueues the message for further processing by the module service procedure, or passes the message to the next module in the stream.

- High-priority messages are typically processed immediately. Immediate processing is not required by the put procedure and the message is passed to the next module.

- Ordinary (or normal) messages are either queued or passed along the stream.

Example 10-2 shows a module write-side put procedure.

EXAMPLE 10-2   Write-side put Procedure

```
static int
modwput (queue_t *q, mblk_t *mp)
{
  struct mod_prv *modptr;

 modptr = (struct mod_prv *) q->q_ptr;  /*state info*/

 if (mp->b_datap->db_type >= QPCTL){ /* proc pri msg and pass it on */
   putnext(q, mp);
   return (0);
  }

 switch(mp->b_datap->db_type) {
  case M_DATA:   /* can process message data queue msg */
   putq(q, mp);  /* for service procedure or pass message */
        /* along with putnext(q,mp) */
   return (0);

  case M_PROTO:
   .
   .
   .

  case M_IOCTL:  /* if cmd in msg is recognized */
      /* process message and send reply back */
      /* else pass message downstream */

  default:
   putnext(q, mp);
   return (0);
  }
}
```

The write-side put procedure, unlike the read side, can be passed M_IOCTL messages. The module must recognize and process the ioctl(2) command, or pass the message downstream if it does not recognize the command.

Example 10-3 shows a general scenario employed by the module's service procedure.

EXAMPLE 10-3 STREAMS Module Service Procedure

```
static int
modrsrv (queue_t *q)
{
  mblk_t *mp;

  while ((mp = getq(q)) != NULL) {
   /* flow control check */
   if (!(mp->b_datap->db_type >= QPCTL) && !canputnext(q)) {
    putbq(q, mp);      /* return message */
    return (0);
   }
   /* process the message */
    .
    .
    .
   putnext(q, mp); /* pass the result */
  }
  return (0);
}
```

The steps are:

1. Retrieve the first message from the queue using `getq`(9F).

2. If the message is high priority, process it immediately and pass it along the stream.

   Otherwise, the `service` procedure should use `canputnext`(9F) to determine if the next module or driver that enqueues messages is within acceptable flow-control limits.

   `canputnext`(9F) searches the stream for the next module, driver, or the stream head with a `service` procedure. When it finds one, it looks at the total message space currently allocated to the queue for messages. If the amount of space currently used at that queue reaches the high-water mark, `canputnext`(9F) returns false (zero). If the next queue with a `service` procedure is within acceptable flow-control limits, `canputnext`(9F) returns true (nonzero).

3. If `canputnext`(9F) returns false, the `service` procedure returns the message to its own queue with `putbq`(9F). The `service` procedure can do no further processing at this time, and it returns to the caller.

   If `canputnext`(9F) returns true, the `service` procedure completes any processing of the message. This can involve retrieving more messages from the queue, allocating and deallocating header and trailer information, and performing control functions for the module.

4. When the `service` procedure is finished processing the message, it calls `putnext`(9F) to pass the resulting message to the next queue.

These steps are repeated until `getq`(9F) returns NULL (the queue is empty) or `canputnext`(9F) returns false.

Filter Module Example

The module shown next, `crmod` in Example 10-4 , is an asymmetric filter. On the write side, a `newline` is changed to a `carriage return` followed by a `newline`. On the read side, no conversion is done.

EXAMPLE 10-4   Filter Module

```
/* Simple filter
 * converts newline -> carriage return, newline
 */
#include <sys/types.h>
#include <sys/param.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

static struct module_info minfo =
 { 0x09, "crmod", 0, INFPSZ, 512, 128 };

static int modopen (queue_t*, dev_t*, int, int, cred_t*);
static int modrput (queue_t*, mblk_t*);
static int modwput (queue_t*, mblk_t*);
static int modwsrv (queue_t*);
static int modclose (queue_t*, int, cred_t*);

static struct qinit rinit = {
 modrput, NULL, modopen, modclose, NULL, &minfo, NULL};

static struct qinit winit = {
 modwput, modwsrv, NULL, NULL, NULL, &minfo, NULL};

struct streamtab crmdinfo={ &rinit, &winit, NULL, NULL};
```

`stropts.h` includes definitions of flush message options common to user applications. `modrput` is like `modput` from the null module.

In contrast to the null module example, a single `module_info` structure is shared by the read side and write side. The `module_info` includes the flow control high-water and low-water marks (512 and 128) for the write queue. (Though the same `module_info` is used on the read queue side, the read side has no `service` procedure so flow control is not used.) The `qinit` contains the `service` procedure pointer.

The write-side `put` procedure, the beginning of the `service` procedure, and an example of flushing a queue are shown in Example 10-5.

EXAMPLE 10-5 Flushing a Queue

```
static int
modwput(queue_t *q, mblk_t *mp)
{
 if (mp->b_datap->db_type >= QPCTL && mp->b_datap->db_type != M_FLUSH)
   putnext(q, mp);
 else
   putq(q, mp);      /* Put it on the queue */
 return (0);
}
static int
modwsrv(queue_t *q)
{
 mblk_t *mp;

 while ((mp = getq(q)) != NULL) {
   switch (mp->b_datap->db_type) {
    default:
     if (canputnext(q)) {
       putnext(q, mp);
       break;
      } else {
       putbq(q, mp);
       return (0);
      }

    case M_FLUSH:
        if (*mp->b_rptr & FLUSHW)
          flushq(q, FLUSHDATA);
        putnext(q, mp);
        break;

    case M_DATA: {
     mblk_t *nbp = NULL;
     mblk_t *next;
     if (!canputnext(q)) {
      putbq(q, mp);
      return (0);
     }
    /* Filter data, appending to queue */
    for (; mp != NULL; mp = next) {
      while (mp->b_rptr < mp->b_wptr) {
        if (*mp->b_rptr == '\n')
         if (!bappend(&nbp, '\r'))
           goto push;
        if (!bappend(&nbp, *mp->b_rptr))
         goto push;
        mp->b_rptr++;
        continue;
       push:
        if (nbp)
         putnext(q, nbp);
        nbp = NULL;
        if (!canputnext(q)) {
         if (mp->b_rptr>=mp->b_wptr){
           next = mp->b_cont;
```

```
        freeb(mp);
        mp=next;
      }
      if (mp)
        putbq(q, mp);
      return (0);
     }
    } /* while */
    next = mp->b_cont;
    freeb(mp);
    if (nbp)
     putnext(q, nbp);
   } /* for */
  }
 } /* switch */
 }
 return (0);
}
```

modwsrv is the write service procedure. It takes a single argument, which is a pointer to the write queue. modwsrv switches on the message type, M_FLUSH or M_DATA. modwsrv processes only one high-priority message, M_FLUSH. No other high-priority messages should reach modwsrv. High-priority messages other than type M_FLUSH use putnext(9F) to avoid scheduling. The others are queued for the service procedure. An M_FLUSH message is a request to remove messages on one or both queues. It can be processed in the put or service procedure.

For an M_FLUSH message, modwsrv checks the first data byte. If FLUSHW is set, the write queue is flushed by flushq(9F), which takes two arguments, the queue pointer and a flag. The flag indicates what should be flushed, data messages (FLUSHDATA) or everything (FLUSHALL). Data includes M_DATA, M_DELAY, M_PROTO, and M_PCPROTO messages. The choice of what types of messages to flush is specific to each module.

Ordinary messages are returned to the queue if canputnext(9F) returns false, indicating the downstream path is blocked.

The differences in M_DATA processing between this and the example in "Message Allocation and Freeing" on page 116 relate to the manner in which the new messages are forwarded and flow controlled. For the purpose of demonstrating alternative means of processing messages, this version creates individual new messages rather than a single message containing multiple message blocks. When a new message block is full, it is immediately forwarded with putnext(9F) rather than being linked into a single large message. This alternative is not desirable because message boundaries are altered, and because of the additional overhead of handling and scheduling multiple messages.

When the filter processing is performed (following push), flow control is checked (with canputnext(9F)) after each new message is forwarded. This is necessary because there is no provision to hold the new message until the queue becomes unblocked. If the downstream

path is blocked, the remaining part of the original message is returned to the queue. Otherwise, processing continues.

Data Flow Control

To support the STREAMS flow control mechanism, modules that use `service` procedures must invoke `canputnext(9F)` before calling `putnext(9F)`, and use appropriate values for the high-water and low-water marks. If your module has a `service` procedure, you manage the flow control. If you don't have a `service` procedure, then there is no need to do anything.

The queue `hiwat` and `lowat` values limit the amount of data that can be placed on a queue. These limits prevent depletion of buffers in the buffer pool. Flow control is advisory in nature and can be bypassed. It is managed by high-water and low-water marks and regulated by the utility routines `getq(9F)`, `putq(9F)`, `putbq(9F)`, `insq(9F)`, `rmvq(9F)`, and `canputnext(9F)`.

The following scenario takes place normally in flow control:

A driver sends data to a module using `putnext(9F)`, and the module's `put` procedure queues data using `putq(9F)`. Calling `putq(9F)` enables the `service` procedure and executes it at some indeterminate time in the future. When the `service` procedure runs, it retrieves the data by calling `getq(9F)`.

If the module cannot process data at the rate at which the driver is sending the data, the following happens:

When the message is queued, `putq(9F)` increments the value of `q_count` by the size of the message and compares the result to the module's high-water limit (`q_hiwat`) value for the queue. If the count reaches `q_hiwat`, `putq(9F)` sets the internal `FULL` indicator for the queue. This causes messages from upstream in the case of a write-side queue or downstream in the case of a read-side queue to be halted (`canputnext(9F)` returns `FALSE`) until the queue count drops below `q_lowat`. `getq(9F)` decrements the queue count. If the resulting count is below `q_lowat`, `getq(9F)` back-enables and causes the `service` procedure to be called for any blocked queue. (Flow control does not prevent reaching `q_hiwat` on a queue. Flow control can exceed its maximum value before `canputnext(9F)` detects `QFULL` and flow is stopped.)

The next example show a line discipline module's flow control. Example 10-6 shows a read-side line discipline module and a write-side line discipline module. Note that the read side is the same as the write side but without the `M_IOCTL` processing.

EXAMPLE 10-6 Read-side Line Discipline Module

```
/* read side line discipline module flow control */
static mblk_t *read_canon(mblk_t *);

static int
ld_read_srv(
 queue_t *q)        /* pointer to read queue */
{
 mblk_t *mp;        /* original message */
 mblk_t *bp;        /* canonicalized message */

 while ((mp = getq(q)) != NULL) {
   switch (mp->b_datap->db_type) { /* type of msg */
   case M_DATA:  /* data message */
    if (canputnext(q)) {
      bp = read_canon(mp);
      putnext(q, bp);
    } else {
      putbq(q, mp); /* put messagebackinqueue */
      return (0);
    }
    break;

   default:
    if (mp->b_datap->db_type >= QPCTL)
      putnext(q, mp);   /* high-priority message */
    else { /* ordinary message */
      if (canputnext(q))
         putnext(q, mp);
      else {
         putbq(q, mp);
         return (0);
      }
    }
    break;
   }
 }
return (0);
}

/* write side line discipline module flow control */
static int
ld_write_srv(
 queue_t *q)         /* pointer to write queue */
{
 mblk_t *mp;         /* original message */
 mblk_t *bp;         /* canonicalized message */

 while ((mp = getq(q)) != NULL) {
   switch (mp->b_datap->db_type) { /* type of msg */
   case M_DATA:     /* data message */
    if (canputnext(q)) {
      bp = write_canon(mp);
      putnext(q, bp);
    } else {
      putbq(q, mp);
```

```
      return (0);
    }
    break;

   case M_IOCTL:
    ld_ioctl(q, mp);
    break;

   default:
    if (mp->b_datap->db_type >= QPCTL)
      putnext(q, mp);  /* high priority message */
    else {        /* ordinary message */
      if (canputnext(q))
        putnext(q, mp);
      else {
        putbq(q, mp);
        return (0);
      }
    }
    break;
   }
 }
 return (0);
 }
```

Design Guidelines

Module developers should follow these guidelines:

- If a module cannot process the message types, the message types must be passed to the next module.

- The module that acts on an M_IOCTL message should send an M_IOCACK or M_IOCNAK message in response to the ioctl(2). If the module cannot process the ioctl(2), it should pass the M_IOCTL message to the next module.

- Modules should not pertain to any particular driver but should be compatible with all drivers.

- In general, modules should not require the data in an M_DATA message to follow a particular format, such as a specific alignment. This means modules can be arbitrarily pushed on top of each other in a sensible fashion. Not following this rule can limit module usability.

- Filter modules pushed between a service user and a service provider should not alter the contents of the M_PROTO or M_PCPROTO block in messages. The contents of the data blocks can be changed, but the message boundaries must be preserved.

`htonl(3B)` and `ntohl(3B)`

The `htonl`(3SOCKET) and `ntohl`(3SOCKET) conversion routines follow the XNS5 publications. The functions continue to convert 32-bit quantities between network byte order and host byte order.

♦♦♦ **C H A P T E R   1 1**

11

Configuring STREAMS Drivers and Modules

This chapter contains information about configuring STREAMS drivers and modules into the Oracle Solaris operating environment. It describes how to configure a driver and a module for the STREAMS framework only. For more in-depth information on the general configuration mechanism, see "Writing Device Drivers for Oracle Solaris 11.2 ".

This chapter has the following organization:

Kernel Data Structures

The following sections contain descriptions of the pointer relationships maintained by the kernel and the various data structures used in STREAMS drivers. When the kernel accesses a driver, it uses a sequence of pointers in various data structures. It looks first at the data structure relationship, and then the entry point interface for loading the driver into the kernel and accessing the driver from the application level.

The order of data structures the kernel uses to get to a driver is as follows:

modlinkage(9S)   Contains the revision number and a list of drivers to dynamically load. It is used by mod_install in the _init routine to load the module into the kernel. Points to a modldrv(9S) or modlstrmod(9S).

modldrv(9S)   Contains information about the driver being loaded and points to the devops structure.

modlstrmod(9S)   Points to an fmodsw(9S) structure (which points to a streamtab(9S)). Only used by STREAMS modules.

dev_ops(9S)   Contains list of entry points for a driver, such as attach, and info. Also points to a cb_ops(9S) structure.

cb_ops(9S)   Points to list of threadable entry points to driver, like open, close, read,
                    write, ioctl. Also points to the streamtab.

streamtab(9S)   Points to the read and write queue init structures.

qinit(9S)   Points to the entry points of the STREAMS portion of the driver, such
                    as put, srv, open, close, as well as the mod_info structure. These entry
                    points only process messages.

Each STREAMS driver or module contains the linkage connections for the various data structures which is a list of pointers to dev_ops(9S) structures. In each dev_ops(9S) structure is a pointer to the cb_ops(9S) structure. In the cb_ops(9S) structure contains a pointer to the streamtab struct. If the driver is not a STREAMS driver, streamtab is NULL. If the driver is a STREAMS driver, streamtab contains initialization routines for the driver.

modlinkage

The definition of modlinkage(9S) is:

```
struct modlinkage {
    int     ml_rev;          /* rev of loadable modules system */
    void    *ml_linkage[4];  /* NULL terminated list of linkage
                              * structures */
};
```

modldrv

The definition of modldrv(9S) is:

```
struct modldrv {
  struct mod_ops   *drv_modops;
  char             *drv_linkinfo;
  struct dev_ops   *drv_dev_ops;
};
```

modlstrmod

The definition of modlstrmod(9S) is below. It does not point to dev_ops(9S) structures because modules can only be pushed onto an existing stream.

```
struct modlstrmod {
  struct mod_ops     *strmod_modops;
```

```
  char              *strmod_linkinfo;
  struct fmodsw     *strmod_fmodsw;
};
```

dev_ops

The dev_ops(9S) structure represents a specific class or type of device. Each dev_ops(9S) structure represents a unique device to the operating system. Each device has its own dev_ops(9S) structure, which in turn contains a cb_ops(9S) structure.

```
struct dev_ops  {
  int       devo_rev;                 /* Driver build version */
  int       devo_refcnt;              /* device reference count */
  int       (*devo_getinfo)(dev_info_t *dip, ddi_info_cmd_t infocmd,
            void *arg, void **result);
  int       (*devo_identify)(dev_info_t *dip);
  int       (*devo_probe)(dev_info_t *dip);
  int       (*devo_attach)(dev_info_t *dip, ddi_attach_cmd_t cmd);
  int       (*devo_detach)(dev_info_t *dip, ddi_detach_cmd_t cmd);
  int       (*devo_reset)(dev_info_t *dip, ddi_reset_cmd_t cmd);
  struct cb_ops     *devo_cb_ops;    /* cb_ops ptr for leaf driver*/
  struct bus_ops    *devo_bus_ops;   /* ptr for nexus drivers */
};
```

cb_ops

The cb_ops(9S) structure is the SunOS 5 version of the cdevsw and bdevsw tables of previous versions of UNIX System V. It contains character and block device information and the driver entry points for non-STREAMS drivers.

```
struct cb_ops  {
  int  *cb_open)(dev_t *devp, int flag, int otyp,
       cred_t *credp);
  int  (*cb_close)(dev_t dev, int flag, int otyp,
       cred_t *credp);
  int  (*cb_strategy)(struct buf *bp);
  int  (*cb_print)(dev_t dev, char *str);
  int  (*cb_dump)(dev_t dev, caddr_t addr,daddr_t blkno,
       int nblk);
  int  (*cb_read)(dev_t dev, struct uio *uiop, cred_t *credp);
  int  (*cb_write)(dev_t dev, struct uio *uiop, cred_t *credp);
  int  (*cb_ioctl)(dev_t dev, int cmd, int arg, int mode,
       cred_t *credp, int *rvalp);
  int  (*cb_devmap)(dev_t dev, dev_info_t *dip,
       ddi_devmap_data_t *dvdp, ddi_devmap_cmd_t cmd,
       off_t offset, unsigned int len, unsigned int prot,
       cred_t *credp);
  int  (*cb_mmap)(dev_t dev, off_t off, int prot);
  int  (*cb_segmap)(dev_t dev, off_t off, struct as *asp,
```

```
        caddr_t *addrp, off_t len, unsigned int prot,
        unsigned int maxprot, unsigned int flags,
        cred_t *credp);
  int  (*cb_chpoll)(dev_t dev, short events, int anyyet,
        short *reventsp, struct pollhead **phpp);
  int  (*cb_prop_op)(dev_t dev, dev_info_t *dip,
        ddi_prop_op_t prop_op, int mod_flags, char *name,
        caddr_t valuep, int *length);

     struct streamtab *cb_str;  /* streams information */

  /*
   * The cb_flag fields are here to tell the system a bit about
   * the device. The bit definitions are in <sys/conf.h>.
   */
     int  cb_flag;     /* driver compatibility flag */
};
```

streamtab

The streamtab(9S) structure contains pointers to the structures that hold the routines that actually initialize the reading and writing for a module.

If streamtab is NULL, there are no STREAMS routines and the entire driver is treated as though it was a non-STREAMS driver. The streamtab(9S) indirectly identifies the appropriate open, close, put, service, and administration routines. These driver and module routines should generally be declared static.

```
struct streamtab {
  struct qinit     *st_rdinit;      /* defines read queue */
  struct qinit     *st_wrinit;      /* defines write queue */
  struct qinit     *st_muxrinit;    /* for multiplexing */
  struct qinit     *st_muxwinit;    /* drivers only */
};
```

qinit

The qinit(9S) structure (also shown in Appendix A) contains pointers to the STREAMS entry points. These routines are called by the module-loading code in the kernel.

```
struct qinit {
  int        (*qi_putp)();            /* put procedure */
  int        (*qi_srvp)();            /* service procedure */
  int        (*qi_qopen)();           /* called on each open or push*/
  int        (*qi_qclose)();          /* called on last close or pop*/
  int        (*qi_qadmin)();          /* reserved for future use */
  struct module_info    *qi_minfo;    /* info struct */
  struct module_stat    *qi_mstat;    /* stats struct (opt)*/
};
```

STREAMS Driver Entry Points

As described in Chapter 9, "STREAMS Drivers" and as seen in the previous data structures, there are four STREAMS driver entry points:

load kernel module	_init(9E), _fini(9E), _info(9E)
dev_ops	attach(9E), getinfo(9E)
cb_ops	open(9E), close(9E), read(9E), write(9E), ioctl(9E)
streamtab	put(9E), srv(9E)

`pts` Example

The following real example was taken from the Oracle Solaris operating environment. The driver pts(7D) is the pseudo terminal slave driver.

EXAMPLE 11-1 Stream Pseudo Terminal Module

```
/*
 * Slave Stream Pseudo Terminal Module
 */

#include <sys/types.h>
#include <sys/param.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/stat.h>
#include <sys/errno.h>
#include <sys/debug.h>
#include <sys/cmn_err.h>
#include <sys/modctl.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

static int ptsopen (queue_t*, dev_t*, int, int, cred_t);
static int ptsclose (queue_t*, int, cred_t*);
static int ptswput (queue_t*, mblk_t*);
static int ptsrsrv (queue_t*);
static int ptswsrv (queue_t*);

static int pts_devinfo(dev_info_t *dip, ddi_info_cmd_t infocmd,
        void *arg,void **result);

static struct module_info pts_info = {
  0xface,
```

```
  "pts",
  0,
  512,
  512,
  128
};

static struct qinit ptsrint = {
  NULL,
  ptsrsrv,
  ptsopen,
  ptsclose,
  NULL,
  &pts_info,
  NULL
};

static struct qinit ptswint = {
  ptswput,
  ptswsrv,
  NULL,
  NULL,
  NULL,
  &pts_info,
  NULL
};

static struct streamtab ptsinfo = {
  &ptsrint,
  &ptswint,
  NULL,
  NULL
};

static int pts_identify(dev_info_t *devi);
static int pts_attach(dev_info_t *devi, ddi_attach_cmd_t cmd);
static int pts_detach(dev_info_t *devi, ddi_detach_cmd_t cmd);
static dev_info_t *pts_dip;     /* private copy of devinfo ptr */

extern kmutex_t pt_lock;
extern pt_cnt;
static struct cb_ops cb_pts_ops = {
    nulldev,        /* cb_open */
    nulldev,        /* cb_close */
    nodev,          /* cb_strategy */
    nodev,          /* cb_print */
    nodev,          /* cb_dump */
    nodev,          /* cb_read */
    nodev,          /* cb_write */
    nodev,          /* cb_ioctl */
    nodev,          /* cb_devmap */
    nodev,          /* cb_mmap */
    nodev,          /* cb_segmap */
    nochpoll,       /* cb_chpoll */
    ddi_prop_op,    /* cb_prop_op */
    &ptsinfo,       /* cb_stream */
    D_MP            /* cb_flag */
};
```

```
static struct dev_ops pts_ops = {
   DEVO_REV,      /* devo_rev */
   0,             /* devo_refcnt */
   pts_devinfo,   /* devo_getinfo */
   pts_identify,  /* devo_identify */
   nulldev,       /* devo_probe */
   pts_attach,    /* devo_attach */
   pts_detach,    /* devo_detach */
   nodev,         /* devo_reset */
   &cb_pts_ops,   /* devo_cb_ops */
   (struct bus_ops*) NULL   /* devo_bus_ops */
};

/*
 * Module linkage information for the kernel.
 */

static struct modldrv modldrv = {
   &mod_driverops,   /* Type of module: a pseudo driver */
   "Slave Stream Pseudo Terminal driver'pts'",
   &pts_ops,     /* driver ops */
};

static struct modlinkage modlinkage = {
  MODREV_1,
  (void *)&modldrv,
  NULL
};

int
_init(void)
{
  return (mod_install(&modlinkage));
}

int
_fini(void)
{
  return (mod_remove(&modlinkage));
}

int
_info(struct modinfo *modinfop)
{
  return (mod_info(&modlinkage, modinfop));
}

static int
pts_identify(dev_info_t *devi)
{
  if (strcmp(ddi_get_name(devi), "pts") == 0)
   return (DDI_IDENTIFIED);
  else
   return (DDI_NOT_IDENTIFIED);
}

static int
```

```
pts_attach(dev_info_t *devi, ddi_attach_cmd_t cmd)
{
  int i;
  char name[5];

  if (cmd != DDI_ATTACH)
   return (DDI_FAILURE);

  for (i = 0; i < pt_cnt; i++) {
   (void) sprintf(name, "%d", i);
   if (ddi_create_minor_node(devi, name, S_IFCHR, i, NULL, 0)
      == DDI_FAILURE) {
    ddi_remove_minor_node(devi, NULL);
    return (DDI_FAILURE);
   }
  }
  return (DDI_SUCCESS);
}

static int
pts_detach(dev_info_t *devi, ddi_detach_cmd_t cmd)
{
  ddi_remove_minor_node(devi, NULL);
  return (DDI_SUCCESS);
}

static int
pts_devinfo (dev_info_t *dip, ddi_info_cmd_t infocmd, void *arg,
      void **result)
{
  int error;

  switch (infocmd)   {
   case DDI_INFO_DEVT2DEVINFO:
    if (pts_dip == NULL) {
     error = DDI_FAILURE;
    } else {
     *result = (void *) pts_dip;
     error = DDI_SUCCESS;
    }
    break;
   case DDI_INFO_DEVT2INSTANCE:
    *result = (void *) 0;
    error = DDI_SUCCESS;
    break;
   default:
    error = DDI_FAILURE;
  }
  return (error);
}

/* the open, close, wput, rsrv, and wsrv routines are presented
 * here solely for the sake of showing how they interact with the
 * configuration data structures and routines. Therefore, the
 * bulk of their code is not included.
 */
static int
ptsopen(rqp, devp, oflag, sflag, credp)
```

```
  queue_t *rqp;  /* pointer to the read side queue */
  dev_t   *devp;   /* pointer to stream tail's dev */
  int  oflag;    /* the user open(2) supplied flags */
  int  sflag;    /* open state flag */
  cred_t  *credp;  /* credentials */
{
  qprocson(rqp);
  return (0);
}

static int
ptsclose(rqp, flag, credp)
  queue_t  *rqp;
  int    flag;
  cred_t  *credp;
{
  qprocsoff(rqp);
  return (0);
}

static int
ptswput(qp, mp)
  queue_t  *qp;
  mblk_t  *mp;
{
  return (0);
}

static int
ptsrsrv(qp)
  queue_t *qp;
{
  return (0);
}

static int
ptswsrv(qp)
  queue_t  *qp;
{
  return (0);
}
```

STREAMS Module Configuration

The following example shows the structures you need if you are working with a module instead
of a driver. Notice that a modlstrmod(9S) is used in modlinkage(9S), and fmodsw(9S)
points to streamtab(9S) instead of going through dev_ops(9S).

EXAMPLE 11-2 Module Structures

```
extern struct streamtab pteminfo;
```

```
static struct fmodsw fsw = {
  "ptem",
  &pteminfo,
  D_NEW | D_MP
};

/*
 * Module linkage information for the kernel.
 */
extern struct mod_ops mod_strmodops;

static struct modlstrmod modlstrmod = {
  &mod_strmodops,
  "pty hardware emulator",
  &fsw
};

static struct modlinkage modlinkage = {
  MODREV_1,
  (void *)&modlstrmod,
  NULL
};
```

Compilation

Below are some compile, assemble, and link lines for an example driver with two C source files
(example_one.c and example_two.c) and an assembly language source file (example_asm.s).

```
cc -D_KERNEL -c example_one.c
cc -D_KERNEL -c example_two.c
as -P -D_ASM -D_KERNEL -I. -o example_asm.o example_asm.s
ld -r -o example example_one.o example_two.o example_asm.o
```

Kernel Loading

See "Writing Device Drivers for Oracle Solaris 11.2 " for more information on the sequence of
installing and loading device drivers. The basic procedure is to copy your driver to /kernel/
drv and your module to /kernel/strmod. For drivers run add_drv(1M).

Note - The autoload facility looks for modules to reside in /kernel/strmod. If the object
resides elsewhere the module will not be loaded.

Checking the Module Type

Below is sample code that enables a driver to determine if it is running as a regular driver, a module, or a cloneable driver. The open routine returns `sflag`, which is checked.

```
if (sflag == MODOPEN)
   /* then the module is being pushed */
 else if (sflag == CLONEOPEN)
   /* then its being opened as a clonable driver */
 else
   /* its being opened as a regular driver */
```

Tunable Parameters

Certain system parameters referred to by STREAMS are configurable when building a new operating system (see the file `/etc/system` and the "STREAMS Parameters" in "Oracle Solaris 11.2 Tunable Parameters Reference Manual "for further details). These parameters are:

nstrpush
: Maximum number (should be at least 8) of modules that can be pushed onto a single stream.

strmsgsz
: Maximum number of bytes of information that a single system call can pass to a stream to be placed into the data part of a message (in `M_DATA` blocks). Any `write`(2) exceeding this size is broken into multiple messages. A `putmsg`(2) with a data part exceeding this size fails with `ERANGE`. If `STRMSGSZ` is set to 0, the number of bytes passed to a stream is infinite.

strctlsz
: Maximum number of bytes of information that a single system call can pass to a stream to be placed into the control part of a message (in an `M_PROTO` or `M_PCPROTO` block). A `putmsg`(2) with a control part that exceeds this size fails with `ERANGE`.

STREAMS Administrative Driver

The `autopush`(1M) facility configures the list of modules for a STREAMS device. It automatically pushes a prespecified list (`/etc/iu.ap`) of modules onto the stream when the STREAMS device is opened and the device is not already open.

The STREAMS Administrative Driver (SAD) (see the `sad`(7D) man page) provides an interface to the `autopush` mechanism. System administrators can open the SAD driver and set or get `autopush` information on other drivers. The SAD driver caches the list of modules

to push for each driver. When the driver is opened the stream head checks the SAD's cache to determine if the device is configured to have modules pushed automatically. If an entry is found, the modules are pushed. If the device has been opened but not closed, another open does not cause the list of the prespecified modules to be pushed again.

Three options configure the module list:

- Configure for each minor device (that is, a specific major and minor device number)
- Configure for a range of minor devices within a major device
- Configure for all minor devices within a major device

In addition, when configuring the module list, an optional anchor can be placed within the module list. See "STREAMS Anchors" on page 222 for more information.

When the module list is cleared, a range of minor devices has to be cleared as a range and not in parts.

Application Interface

The SAD driver is accessed through the `/dev/sad/admin` or `/dev/sad/user` node. After the device is initialized, a program can perform any autopush configuration. The program should open the SAD driver, read a configuration file to find out what modules need to be configured for which devices, format the information into `strapush` structures, and make the `SAD_SAP` `ioctl(2)` calls. See the `sad(7D)` man page for more information.

All autopush operations are performed through `SAD_SAP` `ioctl(2)` commands to set or get `autopush` information. Only the root user can set `autopush` information, but any user can get the `autopush` information for a device.

The `SAD_SAP` ioctl is a form of `ioctl(fd, cmd, arg)`, where *fd* is the file descriptor of the SAD driver, *cmd* is either `SAD_SAP` (set autopush information) or `SAD_GAP` (get autopush information), and *arg* is a pointer to the structure `strapush`.

The `strapush` structure is shown in the following example:

EXAMPLE 11-3   `strapush` Structure

```
/*
 * maximum number of modules that can be pushed on a
 * stream using the autopush feature should be no greater
 * than nstrpush
 */
#define MAXAPUSH 8

/* autopush information common to user and kernel */

struct apcommon {
```

```
   uint    apc_cmd;         /* command - see below */
   major_t apc_major;       /* major device number */
   minor_t apc_minor;       /* minor device number */
   minor_t apc_lastminor;   /* last minor dev # for range */
   uint    apc_npush;       /* number of modules to push */
};

/* ap_cmd - various options of autopush */
#define SAP_CLEAR      0 /* remove configuration list */
#define SAP_ONE        1 /* configure one minor device */
#define SAP_RANGE      2 /* config range of minor devices */
#define SAP_ALL        3 /* configure all minor devices */

/* format of autopush ioctls */
struct strapush {
  struct apcommon sap_common;
  char sap_list[MAXAPUSH] [FMNAMESZ + 1]; /* module list */
};

#define sap_cmd           sap_common.apc_cmd
#define sap_major         sap_common.apc_major
#define sap_minor         sap_common.apc_minor
#define sap_lastminor     sap_common.apc_lastminor
#define sap_npush         sap_common.apc_npush
```

A device is identified by its major device number, sap_major. The SAD_SAP ioctl(2) has the following options:

SAP_ONE   Configures a single minor device, sap_minor, of a driver

SAP_RANGE   Configures a range of minor devices from sap_minor to sap_lastminor, inclusive

SAP_ALL   Configures all minor devices of a device

SAP_CLEAR   Clears the previous settings by removing the entry with the matching sap_major and sap_minor fields

The list of modules is specified as a list of module names in sap_list. MAXAPUSH defines the maximum number of modules to push automatically.

A user can query the current configuration status of a given major/minor device by issuing the SAD_GAP ioctl(2) with sap_major and sap_minor values of the device set. On successful return from this system call, the strapush structure is filled in with the corresponding information for the device. The maximum number of entries that the SAD driver can cache is determined by the tunable parameter NAUTOPUSH which is found in the SAD driver's master file.

The following is an example of an autopush configuration file in /etc/iu.ap:

```
# major     minor        lastminor     modules

 wc         0            0             ldterm ttcompat
 zs         0            1             ldterm ttcompat
```

```
ptsl     0          15           ldterm ttcompat
```

The first line configures a single minor device whose major name is `wc`. Minor numbers start and end at 0, creating only one minor number. The modules automatically pushed are `ldterm` and `ttcompat`. The second line configures the `zs` driver whose minor device numbers are 0 and 1, and automatically pushes the same modules. The last line configures the `ptsl` driver whose minor device numbers are from 0 to 15, and automatically pushes the same modules.

STREAMS Anchors

An anchor is a lock that prevents the removal of a STREAMS module with an `I_POP` call. You place an anchor in a stream on the module you want to lock. All modules at or below the anchor are locked, and can only be popped by a privileged process.

prevents the removal of a STREAMS module with an `I_POP` call. You place an anchor in a stream on the module you want to lock. All modules at or below the anchor are locked, and can only be popped by a *privileged* process.

Anchors and Data Flow

Note - *Hardening Information.* Anchors do not affect the flow of data in the stream or any other properties of the stream other than to lock down its plumbing. Any process can place an anchor on a stream, but once placed, it can only be removed by a privileged process.

An anchor is a per-stream entity; that is, there is exactly one per stream, and the anchor is moved upstream or downstream as needed. When a stream is created, the anchor is conceptually at the driver and therefore has no effect on the stream. By issuing the `I_ANCHOR` ioctl on a stream, a process places the anchor at the STREAMS module directly below the stream head. This means that a process can move an existing anchor upstream by pushing additional STREAMS modules and calling `I_ANCHOR` again.

Although anchors conceptually exist at a specific location in the stream, they are not a data processing element and therefore do not physically exist in the stream (for example, you will not find them parsing `q_next` pointers.) This means that anchors will not appear in `ioctls` such as `I_LOOK`, and they are not included in the module count on the stream.

To remove an anchor, a process `pops` the module at which the anchor was placed. The anchor will only allow a privileged process to pop modules at or below it, which provides security. Once an anchor has been removed, the anchor is not reset to its previous location in the stream, but rather positioned at the STREAMS driver again. When an unprivileged process attempts to `pop` an anchored module, the `ioctl` returns with `EPERM`.

The `I_ANCHOR` ioctl is processed completely in the stream head, and is never sent downstream. If a module or driver sends an `I_ANCHOR` to the stream head, the anchor is silently discarded.

Using Anchors

An anchor can be placed on a STREAMS module by adding an `[anchor]` flag to an `autopush` configuration file or by directly calling the `I_ANCHOR` ioctl.

For example, this configuration file specifies that `autopush` should place an anchor between `foo` and `babar` in the `bb` stream:

```
# major      minor      lastminor      modules

 aa          0          0              foo babar
 bb          0          1              foo [anchor] babar
 bb          131072     131073         foo [anchor] babar
```

The following two examples illustrate the use of anchors in a client/server setting in which file descriptors are being passed. They call the `I_ANCHOR` ioctl directly.

In this example, the server program, `fd_server.c`, opens a stream, pushes modules on to it, and places an anchor on `rlmod`. The client program, `fd_client.c` attempts to pop modules, but can only pop `rlmod` or any modules below it if the client is run as root. That is, if the client is run as non-root, the `I_POP` fails.

This example also shows that once the module with the anchor on it is popped by the privileged root process, the anchor is destroyed (technically, it is moved back to the driver, where it has no effect). Subsequent attempts by the client to pop modules will succeed, even if the client is run as non-root.

Finally, this example also illustrates the effect of passing file descriptors, rather than copying modules or the stream as a whole. Specifically, because the stream is not duplicated, all instances of the client operate on the same stream. In this case, running the client repeatedly causes it to work down the list of modules, popping each one off in turn, until all modules have been removed from the stream.

EXAMPLE 11-4 STREAMS Anchors `fd_server.c`

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <fcntl.h>
#include <stdio.h>
#include <stropts.h>
#include <sys/conf.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdlib.h>
```

```
#define SPIPE_PATH "/tmp/mypipe"

int
main(void)
{
 int    pipefd[2];
 struct strrecvfd strrecvfd;
 int    streamfd;

 /*
  * Open a stream to hand back to the client.  Since this
  * is just an example, we don't really care what we open;
  * make a rlmod<->udp<->ip stream.  Stick an anchor above
  * rlmod so the client cannot I_POP rlmod unless it's root.
  */
 streamfd = open("/dev/udp", O_RDWR);
 if (streamfd == -1) {
  perror("open");
  return (EXIT_FAILURE);
 }

 if (ioctl(streamfd, I_PUSH, "rlmod") == -1) {
  perror("ioctl (I_PUSH) rlmod");
  return (EXIT_FAILURE);
 }

 if (ioctl(streamfd, I_ANCHOR, 0) == -1) {
  perror("ioctl (I_ANCHOR)");
  return (EXIT_FAILURE);
 }

 /*
  * Open ourselves for business by making a mounted stream.
  */
 if (pipe(pipefd) == -1) {
  perror("pipe");
  return (EXIT_FAILURE);
 }

 if (ioctl(pipefd[1], I_PUSH, "connld") == -1) {
  perror("ioctl (I_PUSH) connld");
  return (EXIT_FAILURE);
 }

 (void) umask(0);
 (void) close(creat(SPIPE_PATH, 0666));

 if (fattach(pipefd[1], SPIPE_PATH) == -1) {
  perror("fattach");
  return (EXIT_FAILURE);
 }

 /*
  * Accept clients (iterative server)
  */
 for (;;) {
```

```
   if (ioctl(pipefd[0], I_RECVFD, &strrecvfd) == -1) {
    perror("ioctl (I_RECVFD)");
    return (EXIT_FAILURE);
   }

   /*
    * Send the STREAMS descriptor back to the client.
    */
   if (ioctl(strrecvfd.fd, I_SENDFD, streamfd) == -1) {
    perror("ioctl (I_SENDFD)");
    return (EXIT_FAILURE);
   }
  }
}
```

EXAMPLE 11-5   STREAMS Anchors `fd_client.c`

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <fcntl.h>
#include <stdio.h>
#include <stropts.h>
#include <sys/conf.h>
#include <unistd.h>
#include <stdlib.h>

#define SPIPE_PATH "/tmp/mypipe"

int
main(void)
{
 int    serverfd;
 struct strrecvfd strrecvfd;

 /*
  * Open a connection to the server.
  */
 serverfd = open(SPIPE_PATH, O_RDWR);
 if (serverfd == -1) {
  perror("open");
  return (EXIT_FAILURE);
 }

 /*
  * Receive the STREAMS descriptor from the server.
  */
 if (ioctl(serverfd, I_RECVFD, &strrecvfd) == -1) {
  perror("ioctl (I_RECVFD)");
  return (EXIT_FAILURE);
 }

 (void) printf("received the STREAMS descriptor; attempting to pop "
     "the top module\n");

 /*
```

```
 * Try to remove the top module from the stream.
 */
if (ioctl(strrecvfd.fd, I_POP, 0) == -1)
 perror("ioctl (I_POP)");

(void) printf("modules on the stream: ");
(void) fflush(stdout);

/*
 * Print out what the stream currently looks like.
 */
(void) dup2(strrecvfd.fd, 0);
(void) system("strconf | paste -s -d' ' -");

 return (EXIT_SUCCESS);
}
```

◆◆◆ **C H A P T E R 1 2**

12

Multithreaded STREAMS

This chapter describes how to write a multithreaded STREAMS driver or module. It covers the necessary conversion topics so that new and existing STREAMS modules and drivers run in the multithreaded kernel. It describes STREAMS-specific multithreading issues and techniques. For general information, see "Writing Device Drivers for Oracle Solaris 11.2 ".

This chapter contains the following information:

- "Multithreaded (MT) STREAMS Overview" on page 227
- "MT STREAMS Framework" on page 228
- "MT STREAMS Perimeters" on page 230
- "MT SAFE Modules and Drivers" on page 234
- "Routines Used Inside a Perimeter" on page 236
- "MT SAFE Modules Using Explicit Locks" on page 240
- "Preparing to Port" on page 241
- "Sample Multithreaded Device Driver Using a Per Module Inner Perimeter" on page 242
- "Sample Multithreaded Module With Outer Perimeter" on page 249

Multithreaded (MT) STREAMS Overview

The SunOS 5 operating system is fully multithreaded, which means that it can make effective use of the available parallelism of a symmetric shared-memory multiprocessor computer. All kernel subsystems are multithreaded: scheduler, virtual memory, file systems, block/character/ STREAMS I/O, networking protocols, and device drivers.

MT STREAMS requires you to use some different concepts and terminology. These concepts apply not only to STREAMS drivers, but to all device drivers in the Oracle Solaris operating environment. For a more complete description of these terms, see "Writing Device Drivers for Oracle Solaris 11.2 ". Additionally, see Chapter 1, "Overview of STREAMS" of this guide for definitions and Chapter 8, "STREAMS Kernel-Level Mechanisms" for elements of MT drivers.

Some of the multithreaded terms and ideas are.

Thread   Sequence of instructions executed within context of a process.

Lock	Mechanism to restrict access to data structures.
Single Threaded	Restricting access to a single thread.
Multithreaded	Allowing two or more threads access to a data element.
Multiprocessing	Two or more CPUs concurrently executing the Operating System.
Concurrency	Simultaneous execution.
Preemption	Suspending execution for the next thread to run.
Monitor	Portion of code that is single-threaded.
Mutual Exclusion	Exclusive access to a data element by a single thread at one time.
Condition Variables	Kernel event synchronization primitives.
Counting Semaphores	Memory-based synchronization mechanism.
Readers/Writer Locks	Data lock allowing one writer or many readers at one time.
Callback	On occurrence of a specific event, call a module function.
Synchronous Access	Only one thread is allowed in the perimeter. Upon return from a call the action is complete; when the thread is done, the job is done.
Asynchronous Access	Multiple threads are allowed in the perimeter. Upon return form a call there is no guarantee that the job is complete.
Perimeter Claim	A thread has synchronous access in the perimeter. The claim prevents subsequent synchronous access until the claim is released.
Exclusive Access	Calling a synchronous entry point in the perimeter.
Writer	A thread that has exclusive access to a perimeter.

MT STREAMS Framework

The STREAMS framework consists of the stream head, documented STREAMS data structures (such as `queue_t`, `mblk_t`) and STREAMS utility routines including STREAMS facilities documented in the Device Driver Interface (DDI). The STREAMS framework enables multiple kernel threads to concurrently enter and execute code defined by each module, including the `open`, `close`, `put`, and `service` procedures of each queue within the system.

The first goal of the SunOS 5 system is to preserve the interface and flavor of STREAMS and to shield module code as much as possible from the impact of migrating to the multithreaded kernel. Most of the locking is hidden from the programmer and performed by the STREAMS kernel framework. As long as module code uses the standard, documented programmatic interfaces to shared kernel data structures (such as `queue_t`, `mblk_t`, and `dblk_t`), and adheres to the DDI/DKI, the user does not have to explicitly lock these framework data structures.

The second goal is to make writing MT SAFE modules simple. One of the ways that the framework accomplishes this is by using the MT STREAMS perimeter mechanisms for controlling and restricting concurrent access to a STREAMS module. STREAMS perimeters allow the module writer to select the level of concurrency that a module can tolerate.

STREAMS Framework Integrity

The STREAMS framework ensures the integrity of the STREAMS data structures, such as `queue_t`, `mblk_t`, and `dblk_t` as long as the module conforms to the DDI/DKI, and does not directly access global operating system data structures or facilities not described in the DDI/DKI.

The `q_next` fields of the `queue_t` structure are not modified by the framework while a thread is actively executing within a synchronous entry point. However the `q_next` field might change while a thread is executing within an asynchronous entry point.

The `q_ptr` field is considered private to the module and the framework will not manipulate its value. When making a module MT Safe, the integrity of the module-private data structures must be ensured by the module itself. This integrity can be guaranteed by creating private locks, or by the control of concurrency within the module by the use of STREAMS perimeters. Knowing what the framework supports is critical in deciding what the module writer must provide.

Note - *Hardening Information.* As in previous operating environment releases, a module must not call another module's `put` or `service` procedures directly. The DDI/DKI routines putnext(9F), put(9F), and other routines in Section 9F must be used to pass a message to another queue. Calling another module's routines directly circumvents the design of the MT STREAMS framework and can yield unknown results.

Note - *Hardening Information.* Once a message is passed using a `putq`, `put`, `putnext`, as well as the perimeter function `qwriter`, it cannot be accessed again because the use of this message has been given to the new routine. If a reference needs to be retained by the module, it should copy it by using `copyb`, `copymsg`, `dupb`, or `dupmsg`.

Message Ordering

The STREAMS framework guarantees the ordering of messages along a stream if all the modules in the stream preserve message ordering internally. This ordering guarantee only applies to messages that are sent along the same stream and produced by the same source.

The STREAMS framework does not guarantee that a message has been seen by the next `put` procedure when the call to `putnext(9F)` or `qreply(9F)` returns.

MT STREAMS Perimeters

STREAMS uses a facility known as *perimeters* for handling thread concurrency in STREAMS modules. Perimeters allow the module writer to select conditions that will result in exclusive access to a queue, a pair of queues, or all the queues in a module. This makes multithreading issues easier to work with in STREAMS.

Perimeters work somewhat like reader/writer locks, where there can be many readers, but only one writer. A *synchronous access* to the perimeter is similar to holding the writer lock, in that only one thread can be in the perimeter at a time. Synchronous entry will hold a perimeter exclusively until the thread eventually unwinds out of the perimeter (usually when it returns from a `put`/`putnext` or a `qwriter` call that initially invoked the synchronous behavior). While a thread has synchronous access to the perimeter, all other access (synchronous or asynchronous) will be deferred.

An *asynchronous access* is similar to the reader lock, where many threads can be in the perimeter at a time, including the possible recursive entry of a thread previously entering the perimeter. The asynchronous "claim" is not released until the thread winds out of the `put`/`putnext`. Because asynchronous access is similar to the reader lock, any synchronous access will be deferred or blocked until all asynchronous claims are released.

> ⚠️ **Caution -** *Hardening Information*. The perimeter data is private to the STREAMS framework, and should not be modified by the module itself.

STREAMS enables the definition of two synchronous perimeters. One is an inner perimeter, and is used to define synchronous entry points on a queue or a queue pair (the read and write queue's for a specific module instance). It also identifies an outer perimeter, which is made up of all the inner perimeters for all the queues for a specific module.

There is also a special inner perimeter, `PERMOD`, that is similar to the outer perimeter, but does not have the overhead of the outer perimeter. `PERMOD` identifies a single synchronous entry point for all queues for this module. Because `PERMOD` is like a hybrid of an inner perimeter and outer perimeter, the `PERMOD` perimeter cannot have an outer perimeter.

Inner Perimeters

For the most part, the module writer does not need to specify an inner perimeter, as the STREAMS framework automatically creates it for the module. What needs to be specified is the type of perimeter, and the concurrency of the perimeter.

Inner perimeters come in two types:

D_PERQ   Enables synchronous entry to be different between the read queue and the write queue. Therefore, if a synchronous `putnext` is occurring on the read queue, a synchronous or asynchronous `putnext` can occur on the write queue (or other (a)synchronous access on the write queue).

D_QPAIR   Protects both the read queue and the write queue, so synchronous access to one queue will prevent synchronous or asynchronous access to the other queue.

Another perimeter, `D_PERMOD` is slightly different, and is discussed in "PERMOD Perimeter" on page 232.

An inner perimeter becomes exclusive (writer) whenever an inner synchronous entry point is encountered. By default all the entry points are considered to be synchronous until enabled as "shared" entry points. As previously stated, synchronous entry points remain exclusive until the thread returns to the caller of the synchronous entry point. If the synchronous function calls `putnext`, the perimeter remains exclusive across the `putnext`, up till the synchronous function can return to its caller, and subsequent entries into the perimeter will be deferred.

Inner perimeters can specify additional concurrency on the STREAMS entry points for `open`/`close`, `put`/`putnext`, `service`, and `callbacks` as shown in Table 12-4.

Outer Perimeters

The module writer can also specify an outer perimeter. An outer perimeter is the linked list of all inner perimeters for all queues associated with the specified module. Entering the outer perimeter is equivalent to entering each of the inner perimeters. As this can also be an expensive operation, the outer perimeter is *only* entered synchronously, and upon successful completion of a `qwriter(PERIM_OUTER)` makes the outer perimeter exclusive. This also has the effect of making each of the inner perimeters exclusive.

Use of outer perimeters is reserved for module data that has an effect on all queue instances of the module, such as module state that might allow messages to pass between other instances of the module, information that allows a driver to configure shared hardware, or at `open`/`close` time when information is needed for all open instances for a module.

Outer perimeters, at this time, have only one concurrency modifier. This is `D_MTOCEXCL`, and instructs the framework to enter the outer perimeter on each `open` and `close` of queues for the module.

`PERMOD` Perimeter

The `PERMOD` perimeter is a hybrid of the inner and outer perimeter. It is implemented primarily for modules that might have a large number of queue instances, and cannot afford the latency for entering the outer perimeter. Because it is a hybrid, `PERMOD` perimeters cannot have an outer perimeter, and modules that have `D_MTPERMOD` and `D_MTOUTPERIM` defined will fail at `open`. As `PERMOD` perimeters are implemented as inner perimeters, they share all the concurrency states as the inner perimeter, see Table 12-2.

Hot Perimeters

All STREAMS modules and drivers in the Oracle Solaris operating environment must be `D_MTSAFE`, and must account for multithreading. Specifying an inner and/or outer perimeter will handle concurrency issues that the module writer may encounter while developing the module or driver. Experienced STREAMS programmers might decide that the perimeter should not have any synchronous entry points, and should run fully hot. To define a fully-hot perimeter, the module writer need only specify the `D_MTSAFE` flag without an inner perimeter type (`D_MTPERQ`, `D_MTQPAIR`, `D_MTPERMOD`) and without an outer perimeter (`D_MTOUTPERIM`).

⚠️ **Caution -** *Hardening Information*. All STREAMS entry points run concurrently, and in a multiprocessor environment, there can be a `put` procedure running simultaneously with a `service` procedure or even a `close` procedure. So the writer must take precautions against kernel panics by making sure that other concurrent threads will not reference data the current thread is trying to change or remove.

Defining Perimeter Types

To configure a module with perimeter types and concurrency types, use the `f_flag` field in `fmodsw`(9s) with `D_MTSAFE` or'd with the appropriate perimeter type flags. See Table 12-1.

The easiest method is to initially implement your module and configure it to be per-module single threaded, and increase the level of concurrency as needed. "Sample Multithreaded Device Driver Using a Per Module Inner Perimeter" on page 242 provides a complete

example of using a per-module perimeter, and "Sample Multithreaded Module With Outer Perimeter" on page 249 provides a complete example with a higher level of concurrency.

To configure a driver with perimeter and concurrency types, put `MT_SAFE` and the appropriate perimeter flags in the `cb_flag` field of the `cb_ops` structure for the driver.

Choosing a Perimeter Type

Table 12-1 summarizes examples of when to use an inner perimeter, or both an inner and outer perimeter for a STREAMS module.

TABLE 12-1   Choosing a Perimeter Type

Module Description	Perimeter Type
A module where the `put` procedure reads as well as modifies module global data. Use a per-module inner perimeter on the single-threaded module.	inner
A module where all the module private data associated with a queue (or a read/write pair of queues) can be configured to be single-threaded. Use an inner perimeter for each corresponding queue (or queue pair) .	inner
A module where most of the module private data is associated with a queue (or a queue pair); but has some module global data that is mostly read. Use an inner perimeter for the queue (or queue pair) plus an outer perimeter for global data. Use `qwriter` to protect the sections where it modifies the module's global data (see "qwriter" on page 237).	inner and outer
A module that requires higher concurrency for certain message types while not requiring message ordering. Use an inner perimeter for shared access to the `put` procedures. Use an outer perimeter for `put` procedures that require exclusive access. Use `qwriter` when messages are handled in the `put` procedures that require exclusive access (see "qwriter" on page 237).	inner and outer
A hardware driver can use an appropriate set of inner and outer perimeters to restrict the concurrency in the `open`, `close`, `put`, and `service` procedures. With explicit synchronization primitives (`mutex`, condition variables, readers/writer, semaphore), these drivers restrict the concurrency when accessing the hardware registers in interrupt handlers. When designing such drivers, you need to be aware of the issues listed in "MT SAFE Modules Using Explicit Locks" on page 240.	inner and outer

Several flags specify the inner and outer perimeters (see Table 12-2 and Table 12-3). These flags fall into three categories:

■ Define the presence and scope of the inner perimeter
■ Define the presence of the outer perimeter (which can have only one scope)
■ Modify the default concurrency for the different entry points

You configure the inner perimeter by choosing one of the mutually exclusive flags shown in Table 12-2.

TABLE 12-2   Inner Perimeter Flags

Flag	Description
D_MTPERMOD	The module has an inner perimeter that encloses all the module's queues
D_MTAPAIR	The module has an inner perimeter around each read/write pair of queues
D_MTPERQ	The module has an inner perimeter around each queue
none	The module has no inner perimeter

Configure the outer perimeter using the flag shown in Table 12-3.

TABLE 12-3   Outer Perimeter Flag

Flag	Description
D_MTOUTEPERIM	In addition to an inner perimeter (or none), the module has an outer perimeter that encloses all the module's queues. This can be combined with all the inner perimeter options except D_MTPERMOD.

By default all synchronous entry points enter the inner perimeter exclusively and enter the outer perimeter shared. To modify this behavior use the flags shown in Table 12-4.

TABLE 12-4   Modify Exclusive/Shared Access Flags

Flag	Description
D_MTOCEXCL	open/close entry points are synchronous.
D_MTPUTSHARED	put/putnext entry points are asynchronous
_D_MTOCSHARED	open/close entry points are asynchronous (experimental)
_D_MTCBSHARED	callbacks (via qtimeout, qbufcall) are asynchronous (experimental)
_D_MTSVCSHARED	service procedures are asynchronous (experimental)

⚠ **Caution -** *Hardening Information.* Concurrency flags designated with a preceding underbar "_" are experimental, and their behavior might change in the future and should not be relied upon.

MT SAFE Modules and Drivers

A module or a driver can be either MT SAFE or MT UNSAFE. A module or driver is MT SAFE when its data values are correct regardless of the order that multiple threads access

and modify the data. For MT SAFE mode, use MT STREAMS perimeters to restrict the concurrency in a module or driver to:

- Per-module single threading
- Per queue-pair single threading
- Per queue single threading
- Per queue or per queue-pair single threading of the `put` and `service` procedures with per module single threading of the `open` and `close` routines
- Unrestricted concurrency in the `put` and `service` procedures with the ability to restrict the concurrency when handling messages that modify data
- Completely unrestricted concurrency

MT SAFE Module

To configure a module as being MT SAFE, use the `f_flag` field in `fmodsw`(9S).

The easiest method is to initially implement your module and configure it to be per-module single threaded, and increase the level of concurrency as needed. "Sample Multithreaded Device Driver Using a Per Module Inner Perimeter" on page 242 provides a complete example of using a per-module perimeter, and "Sample Multithreaded Module With Outer Perimeter" on page 249 provides a complete example with a higher level of concurrency.

MT SAFE modules can use different MT STREAMS perimeters to restrict the concurrency in the module to a concurrency that is natural given the data structures that the module contains, thereby removing the need for module private locks (see "MT STREAMS Perimeters" on page 230 for information on perimeters). A module that requires unrestricted concurrency can be configured to have no perimeters. Such modules have to use explicit locking primitives to protect their data structures. While such modules can exploit the maximum level of concurrency allowed by the underlying hardware platform, they are more complex to develop and support. See "MT SAFE Modules Using Explicit Locks" on page 240.

Independent of the perimeters, there will be at most one thread allowed within any given queue's service procedure.

Your MT SAFE modules should use perimeters and avoid using module private locks (`mutex`, condition variables, readers/writer, or semaphore). Should you opt to use module private locks, you need to read "MT SAFE Modules Using Explicit Locks" on page 240 along with this section.

Note - MT UNSAFE mode for STREAMS modules was temporarily supported as an aid in porting SVR4 modules; however, MT UNSAFE is not supported after SVR4.

> **Note -** Upper and lower multiplexors share the same perimeter type and concurrency level.

MT SAFE Driver

To configure a driver as being MT SAFE, initialize the cb_ops(9S) and dev_ops(9S) data structures. This code must be in the header section of your module. For more information, see Example 12-1, and dev_ops(9S).

The driver is configured to be MT SAFE by setting the cb_flag to D_MP. It also specifies any MT STREAMS perimeters by setting flags in the cb_flag field. (See mt-streams(9F).)

Routines Used Inside a Perimeter

This section describes the routines and data fields used after you enter a perimeter.

qprocson/qprocsoff

The routines qprocson(9F) and qprocsoff(9F) respectively enable and disable the put and service procedures of the queue pair. Before calling qprocson(9F) and after calling qprocsoff(9F), the module's put and service procedures are disabled; messages flow around the module as if it were not present in the stream.

Call qprocson(9F) in the first open of a module, but only after allocating and initializing any module resources on which the put and service procedures depend. Call the qprocsoff routine in the close routine of the module before deallocating any resources on which the put and service procedures depend.

> **Note -** To avoid deadlocks, modules must not hold private locks across the calls to qprocson(9F) or qprocsoff(9F).

qtimeout/qunbufcall

The timeout(9F) and bufcall(9F) callbacks are asynchronous. For a module using MT STREAMS perimeters, the timeout(9F) and bufcall(9F) callback functions execute outside

the scope of the perimeters. This makes synchronization of callbacks with the rest of the module complex.

To make `timeout`(9F) and `bufcall`(9F) functionality easier to use for modules with perimeters, there are additional interfaces that use synchronous callbacks. These routines are `qtimeout`(9F), `quntimeout`(9F), `qbufcall`(9F), and `qunbufcall`(9F). When using these routines, the callback functions are executed inside the perimeters, and hence have the same concurrency restrictions as the `put` and `service` procedures.

qwriter

Modules can use the `qwriter`(9F) function to upgrade from shared to exclusive access at a perimeter. For example, a module with an outer perimeter can use `qwriter`(9F) in the `put` procedure to upgrade to exclusive access at the outer perimeter. A module where the `put` procedure runs with shared access at the inner perimeter (`D_MTPUTSHARED`) can use `qwriter`(9F) in the `put` procedure to upgrade to exclusive access at the inner perimeter.

Returning from a `qwriter` call does not mean that the `callback` function has executed. If the framework can become exclusive in the `qwriter` call, it will enter the perimeter synchronously, and execute the `callback`. If it cannot, the `callback` will be deferred. It is a good idea for any caller of `qwriter` to immediately return to its caller as there is little that can be accomplished in this thread of execution.

Caution - *Hardening Information.* Do not call `qwriter` with another queue, as `qwriter` assumes that the caller has already made a claim to the perimeter that the queue is associated with (asynchronous entry), and calling another perimeter will cause problems.

Note - `qwriter`(9F) cannot be used in the `open` or `close` procedures. If a module needs exclusive access at the outer perimeter in the `open` and/or `close` procedures, it has to specify that the outer perimeter should always be entered exclusively for `open` and `close` (using `D_MTOCEXCL`).

The STREAMS framework guarantees that all deferred `qwriter`(9F) callbacks associated with a queue have executed before the module's `close` routine is called for that queue.

For an example of a driver using `qwriter`(9F) see Example 12-2.

`qwait`

A module that uses perimeters and must wait in its `open` or `close` procedure for a message from another STREAMS module has to wait outside the perimeters; otherwise, the message would never be allowed to enter its `put` and `service` procedures. This is accomplished by using the `qwait(9F)` interface. See `qwriter(9F)` man page for an example. For information about signal reception during a `close`, see "close Routine" on page 99

Asynchronous Callback Functions

Interrupt handlers and other asynchronous callback functions require special care by the module writer, because they can execute asynchronously to threads executing within the module `open`, `close`, `put`, and `service` procedures.

For modules using perimeters, use `qtimeout(9F)` and `qbufcall(9F)` instead of `timeout(9F)` and `bufcall(9F)`. The `qtimeout` and `qbufcall` callbacks are synchronous and consequently introduce no special synchronization requirements.

Because a thread can enter the module at any time, you must ensure that the asynchronous callback function acquires the proper private locks before accessing private module data structures, and releases these locks before returning. You must cancel any outstanding registered callback routines before the data structures on which the callback routines depend are deallocated and the module closed.

- For hardware device interrupts, this involves disabling the device interrupts.
- Outstanding callbacks from `timeout(9F)` and `bufcall(9F)` must be canceled by calling `untimeout(9F)` and `unbufcall(9F)`.

  The module cannot hold certain private locks across calls to `untimeout(9F)` or `unbufcall(9F)`. These locks are those that the module's `timeout(9F)` or `bufcall(9F)` callback functions acquire. See "MT SAFE Modules Using Explicit Locks" on page 240.

- If outstanding callbacks from `esballoc(9F)` are associated with a particular stream, they must be allowed to complete before the module `close` routine deallocates the private data structures on which they depend.

`close` Race Conditions

Because the callback functions are by nature asynchronous, they can be executing or about to execute at the time the module `close` routine is called. You must cancel all outstanding callback

and interrupt conditions before deallocating those data structures or returning from the `close` routine.

The callback functions scheduled with `timeout(9F)` and `bufcall(9F)` are guaranteed to have been canceled by the time `untimeout(9F)` and `unbufcall(9F)` return. The same is true for `qtimeout(9F)` and `qbufcall(9F)` by the time `quntimeout(9F)` and `qunbufcall(9F)` return. You must also take responsibility for other asynchronous routines, including `esballoc(9F)` callbacks and hardware, as well as software interrupts.

Unloading a Module that Uses `esballoc`

The STREAMS framework prevents a module or driver text from being unloaded while there are `open` instances of the module or driver. If a module does not cancel all callbacks in the last `close` routine, it should not be allowed to be unloaded.

This is an issue mainly for modules and drivers using `esballoc` because `esballoc` callbacks cannot be canceled. Thus, modules and drivers using `esballoc` have to be prepared to handle calls to the `esballoc` callback free function after the last instance of the module or driver has been closed.

Modules and drivers can maintain a semaphore count of outstanding callbacks. They can deny an unload by making the `_fini(9E)` routine return `EBUSY` if there are outstanding callbacks.

Use of the `q_next` Field

The `q_next` field in the `queue_t` structure can be referenced in `open`, `close`, `put`, and `service` procedures as well as the synchronous callback procedures (scheduled with `qtimeout(9F)`, `qbufcall(9F)`, and `qwriter(9F)`). However, the value in the `q_next` field should not be trusted. It is relevant to the STREAMS framework, but may not be relevant to a specific module.

All other module code, such as interrupt routines, `timeout(9F)` and `esballoc(9F)` callback routines, cannot dereference `q_next`. Those routines have to use the "next" version of all functions. For instance, use `canputnext(9F)` instead of dereferencing `q_next` and using `canput(9F)`.

MT SAFE Modules Using Explicit Locks

Although the result is not reliable, you can use explicit locks either instead of perimeters or to augment the concurrency restrictions provided by the perimeters.

> ⚠️ **Caution -** Explicit locks cannot be used to preserve message ordering in a module because of the risk of re-entering the module. Use MT STREAMS perimeters to preserve message ordering.

All four types of kernel synchronization primitives are available to the module writer: mutexes, readers/writer locks, semaphores, and condition variables. Because `cv_wait` implies a context switch, it can only be called from the module's `open` and `close` procedures, which are executed with valid process context. You must use the synchronization primitives to protect accesses and ensure the integrity of private module data structures.

Constraints When Using Locks

When adding locks in a module, observe these constraints:

- Avoid holding module private locks across calls to `putnext`(9F). The module might be re-entered by the same thread that called `putnext`(9F), causing the module to try to acquire a lock that it already holds. This can cause kernel panic.
- Do not hold module private locks, acquired in `put` or `service` procedures, across the calls to `qprocson`(9F) or qprocsoff(9F). Doing this causes deadlock, since `qprocson`(9F) and qprocsoff(9F) wait until all threads leave the inner perimeter.
- Similarly, do not hold locks, acquired in the `timeout`(9F) and `bufcall`(9F) callback procedures, across the calls to `untimeout`(9F) or `unbufcall`(9F). Doing this causes deadlock, because `untimeout`(9F)and `unbufcall`(9F) wait until an already executing callback has completed.

The first restriction deters using module private locks to preserve message ordering. The preferred mechanism is to use MT STREAMS perimeters to preserve message ordering.

Preserving Message Ordering

Module private locks cannot be used to preserve message ordering because they cannot be held across calls to `putnext`(9F) and the other messages that pass routines to other modules. The alternatives for preserving message ordering are:

- Use MT STREAMS perimeters.

■ Pass all messages through the `service` procedures. The `service` procedure can drop the locks before calling `putnext(9F)` or `qreply(9F)`, without reordering messages, because the framework guarantees that at most, one thread will execute in the `service` procedure for a given queue.

Use perimeters to avoid the performance penalty for using `service` procedures.

Preparing to Port

When modifying a STREAMS driver to take advantage of the multithreaded kernel, a level of MT safety is selected according to:

■ The desired degree of concurrency
■ The natural concurrency of the underlying module
■ The amount of effort or complexity required

Much of the effort in conversion is simply determining the appropriate degree of data sharing and the corresponding granularity of locking (see Table 12-1). The actual time spent configuring perimeters and/or installing locks should be much smaller than the time spent in analysis.

To port your module, you must understand the data structures used within your module, as well as the accesses to those data structures. You must fully understand the relationship between all portions of the module and private data within that module, and to use the MT STREAMS perimeters (or the synchronization primitives available) to maintain the integrity of these private data structures.

You must explicitly restrict access to private module data structures as appropriate to ensure the integrity of these data structures. You must use the MT STREAMS perimeters to restrict the concurrency in the module so that the parts of the module that modify private data are single-threaded with respect to the parts of the module that read the same data. (For more information about perimeters, see "MT STREAMS Perimeters" on page 230.) Besides perimeters, you can use the synchronization primitives available (`mutex`, condition variables, readers/writer, semaphore) to explicitly restrict access to module private data appropriate for the operations within the module on that data.

The first step in multithreading a module or driver is to analyze the module, breaking the entire module up into a list of individual operations and the private data structures referenced in each operation. Part of this first step is deciding upon a level of concurrency for the module. Ask yourself which of these operations can be multithreaded and which must be single-threaded. Try to find a level of concurrency that is "natural" for the module and matches one of the available perimeters (or, alternatively, requires the minimal number of locks) , and has a simple and straightforward implementation. Avoid additional unnecessary complexity.

Typical questions to ask are:

- What data structures are maintained within the module?
- What types of accesses are made to each field of these data structures?
- When is each data structure accessed destructively (written) and when is it accessed non-destructively (read)?
- Which operations within the module should be allowed to execute concurrently?
- Is per module single-threading appropriate for the module?
- Is per queue-pair or per queue single-threading appropriate?
- What are the message ordering requirements?

Porting to the SunOS 5 System

When porting a STREAMS module or driver from the SunOS 4 system to the SunOS 5 system, the module should be examined with respect to the following areas:

- The SunOS 5 Device Driver Interface (DDI/DKI)
- The SunOS 5 MT design

For portability and correct operation, each module must adhere to the SunOS DDI/DKI. Several facilities available in previous releases of the SunOS system have changed and can take different arguments, or produce different side effects, or no longer exist in the SunOS 5 system. The module writer should carefully review the module with respect to the DDI/DKI.

Each module that accesses underlying Sun-specific features included in the SunOS 5 system should conform to the Device Driver Interface. The SunOS 5 DDI defines the interface used by the device driver to register device hardware interrupts, access device node properties, map device slave memory, and establish and synchronize memory mappings for DVMA (Direct Virtual Memory Access). These areas are primarily applicable to hardware device drivers. Refer to the Device Driver Interface Specification within the "Writing Device Drivers for Oracle Solaris 11.2 " for details on the SunOS 5 DDI and DVMA.

The kernel networking subsystem in the SunOS 5 system is based on STREAMS. Datalink drivers that used the `ifnet` interface in the SunOS 4 system must be converted to use DLPI for the SunOS 5 system. Refer to the Data Link Provider Interface, Revision 2 specification.

After reviewing the module for conformance to the SunOS 5 DKI and DDI specifications, you should be able to consider the impact of multithreading on the module.

Sample Multithreaded Device Driver Using a Per Module Inner Perimeter

Example 12-1 is a sample multithreaded, loadable, STREAMS pseudo-driver. The driver MT design is the simplest possible based on using a per module inner perimeter. Thus, only one

thread can execute in the driver at any time. In addition, a quntimeout(9F) synchronous callback routine is used. The driver cancels an outstanding qtimeout(9F) by calling quntimeout(9F) in the close routine. See "close Race Conditions" on page 238.

EXAMPLE 12-1 Multithreaded, Loadable, STREAMS Pseudo-Driver

```
/*
 * Example SunOS 5 multithreaded STREAMS pseudo device driver.
 * Using a D_MTPERMOD inner perimeter.
 */

#include  <sys/types.h>
#include  <sys/errno.h>
#include  <sys/stropts.h>
#include  <sys/stream.h>
#include  <sys/strlog.h>
#include  <sys/cmn_err.h>
#include  <sys/modctl.h>
#include  <sys/kmem.h>
#include  <sys/conf.h>
#include  <sys/ksynch.h>
#include  <sys/stat.h>
#include  <sys/ddi.h>
#include  <sys/sunddi.h>

/*
 * Function prototypes.
 */
static   int xxidentify(dev_info_t *);
static   int xxattach(dev_info_t *, ddi_attach_cmd_t);
static   int xxdetach(dev_info_t *, ddi_detach_cmd_t);
static   int xxgetinfo(dev_info_t *,ddi_info_cmd_t,void *,void**);
static   int xxopen(queue_t *, dev_t *, int, int, cred_t *);
static   int xxclose(queue_t *, int, cred_t *);
static   int xxwput(queue_t *, mblk_t *);
static   int xxwsrv(queue_t *);
static   void xxtick(caddr_t);

/*
 * Streams Declarations
 */
static struct module_info xxm_info = {
   99,            /* mi_idnum */
   "xx",          /* mi_idname */
   0,             /* mi_minpsz */
   INFPSZ,        /* mi_maxpsz */
   0,             /* mi_hiwat */
   0              /* mi_lowat */
};

static struct qinit xxrinit = {
  NULL,           /* qi_putp */
  NULL,           /* qi_srvp */
  xxopen,         /* qi_qopen */
```

```
    xxclose,        /* qi_qclose */
    NULL,           /* qi_qadmin */
    &xxm_info,      /* qi_minfo */
    NULL            /* qi_mstat */
};

static struct qinit xxwinit = {
    xxwput,         /* qi_putp */
    xxwsrv,         /* qi_srvp */
    NULL,           /* qi_qopen */
    NULL,           /* qi_qclose */
    NULL,           /* qi_qadmin */
    &xxm_info,      /* qi_minfo */
    NULL            /* qi_mstat */
};

static struct streamtab xxstrtab = {
    &xxrinit,       /* st_rdinit */
    &xxwinit,       /* st_wrinit */
    NULL,           /* st_muxrinit */
    NULL            /* st_muxwrinit */
};

/*
 * define the xx_ops structure.
 */

static     struct cb_ops cb_xx_ops = {
    nodev,            /* cb_open */
    nodev,            /* cb_close */
    nodev,            /* cb_strategy */
    nodev,            /* cb_print */
    nodev,            /* cb_dump */
    nodev,            /* cb_read */
    nodev,            /* cb_write */
    nodev,            /* cb_ioctl */
    nodev,            /* cb_devmap */
    nodev,            /* cb_mmap */
    nodev,            /* cb_segmap */
    nochpoll,         /* cb_chpoll */
    ddi_prop_op,      /* cb_prop_op */
    &xxstrtab,        /* cb_stream */
    (D_NEW|D_MP|D_MTPERMOD) /* cb_flag */
};

static struct dev_ops xx_ops = {
    DEVO_REV,         /* devo_rev */
    0,                /* devo_refcnt */
    xxgetinfo,        /* devo_getinfo */
    xxidentify,       /* devo_identify */
    nodev,            /* devo_probe */
    xxattach,         /* devo_attach */
    xxdetach,         /* devo_detach */
    nodev,            /* devo_reset */
    &cb_xx_ops,       /* devo_cb_ops */
    (struct bus_ops *)NULL /* devo_bus_ops */
};
```

```
/*
 * Module linkage information for the kernel.
 */
static struct modldrv modldrv = {
  &mod_driverops,   /* Type of module. This one is a driver */
  "xx",             /* Driver name */
  &xx_ops,          /* driver ops */
};

static struct modlinkage modlinkage = {
  MODREV_1,
  &modldrv,
  NULL
};

/*
 * Driver private data structure. One is allocated per Stream.
 */
struct xxstr {
  struct  xxstr *xx_next; /* pointer to next in list */
  queue_t  *xx_rq;     /* read side queue pointer */
  minor_t  xx_minor;   /* minor device # (for clone) */
  int   xx_timeoutid;  /* id returned from timeout() */
};

/*
 * Linked list of opened Stream xxstr structures.
 * No need for locks protecting it since the whole module is
 * single threaded using the D_MTPERMOD perimeter.
 */
static struct xxstr      *xxup = NULL;


/*
 * Module Config entry points
 */

_init(void)
{
   return (mod_install(&modlinkage));
}

_fini(void)
{
   return (mod_remove(&modlinkage));
}

_info(struct modinfo *modinfop)
{
   return (mod_info(&modlinkage, modinfop));
}

/*
 * Auto Configuration entry points
 */

/* Identify device. */
```

```
static int
xxidentify(dev_info_t *dip)
{
   if (strcmp(ddi_get_name(dip), "xx") == 0)
   return (DDI_IDENTIFIED);
   else
   return (DDI_NOT_IDENTIFIED);
}

/* Attach device. */
static int
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
   /* This creates the device node. */
   if (ddi_create_minor_node(dip, "xx", S_IFCHR, ddi_get_instance(dip),
    DDI_PSEUDO, CLONE_DEV) == DDI_FAILURE) {
   return (DDI_FAILURE);
   }
   ddi_report_dev(dip);
   return (DDI_SUCCESS);
}

/* Detach device. */
static int
xxdetach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
   ddi_remove_minor_node(dip, NULL);
   return (DDI_SUCCESS);
}

/* ARGSUSED */
static int
xxgetinfo(dev_info_t *dip, ddi_info_cmd_t infocmd, void *arg,
    void **resultp)
{
   dev_t dev = (dev_t) arg;
   int instance, ret = DDI_FAILURE;

   devstate_t *sp;
   state *statep;
   instance = getminor(dev);

   switch (infocmd) {
   case DDI_INFO_DEVT2DEVINFO:
    if ((sp = ddi_get_soft_state(statep,
      getminor((dev_t) arg))) != NULL) {
     *resultp = sp->devi;
     ret = DDI_SUCCESS;
    } else
     *result = NULL;
    break;

   case DDI_INFO_DEVT2INSTANCE:
    *resultp = (void *)instance;
    ret = DDI_SUCCESS;
    break;

   default:
```

```
      break;
    }
    return (ret);
}

static
xxopen(rq, devp, flag, sflag, credp)
   queue_t   *rq;
   dev_t    *devp;
   int    flag;
   int    sflag;
   cred_t   *credp;
{
   struct xxstr *xxp;
   struct xxstr **prevxxp;
   minor_t    minordev;

   /* If this stream already open - we're done. */
   if (rq->q_ptr)
   return (0);

    /* Determine minor device number. */
   prevxxp = & xxup;
   if (sflag == CLONEOPEN) {
   minordev = 0;
   while ((xxp = *prevxxp) != NULL) {
    if (minordev < xxp->xx_minor)
     break;
    minordev++;
    prevxxp = &xxp->xx_next;
   }
   *devp = makedevice(getmajor(*devp), minordev)
   } else
   minordev = getminor(*devp);

   /* Allocate our private per-Stream data structure. */
   if ((xxp = kmem_alloc(sizeof (struct xxstr), KM_SLEEP)) == NULL)
   return (ENOMEM);

   /* Point q_ptr at it. */
   rq->q_ptr = WR(rq)->q_ptr = (char *) xxp;

   /* Initialize it. */
   xxp->xx_minor = minordev;
   xxp->xx_timeoutid = 0;
   xxp->xx_rq = rq;

   /* Link new entry into the list of active entries. */
   xxp->xx_next = *prevxxp;
   *prevxxp = xxp;

   /* Enable xxput() and xxsrv() procedures on this queue. */
   qprocson(rq);

   return (0);
}

static
```

```
xxclose(rq, flag, credp)
   queue_t    *rq;
   int     flag;
   cred_t    *credp;

{
   struct  xxstr  *xxp;
   struct  xxstr  **prevxxp;

   /* Disable xxput() and xxsrv() procedures on this queue. */
   qprocsoff(rq);
   /* Cancel any pending timeout. */
   xxp = (struct xxstr *) rq->q_ptr;
   if (xxp->xx_timeoutid != 0) {
     (void) quntimeout(rq, xxp->xx_timeoutid);
     xxp->xx_timeoutid = 0;
   }
   /* Unlink per-stream entry from the active list and free it. */
   for (prevxxp = &xxup; (xxp = *prevxxp) != NULL;
    prevxxp = &xxp->xx_next)
   if (xxp == (struct xxstr *) rq->q_ptr)
    break;
   *prevxxp = xxp->xx_next;
   kmem_free (xxp, sizeof (struct xxstr));

   rq->q_ptr = WR(rq)->q_ptr = NULL;

   return (0);
}

static
xxwput(wq, mp)
   queue_t   *wq;
   mblk_t   *mp;
{
   struct xxstr *xxp = (struct xxstr *)wq->q_ptr;

  /* write your code here */
     /* *** Sacha's Comments *** broken */
   freemsg(mp);
  mp = NULL;

   if (mp != NULL)
    putnext(wq, mp);
}

static
xxwsrv(wq)
  queue_t   *wq;
{
  mblk_t   *mp;
  struct xxstr *xxp;

  xxp = (struct xxstr *) wq->q_ptr;

  while (mp = getq(wq)) {
   /* write your code here */
    freemsg(mp);
```

```
   /* for example, start a timeout */
   if (xxp->xx_timeoutid != 0) {
    /* cancel running timeout */
    (void) quntimeout(wq, xxp->xx_timeoutid);
   }
   xxp->xx_timeoutid = qtimeout(wq, xxtick, (char *)xxp, 10);
  }
}

static void
xxtick(arg)
  caddr_t arg;
{
  struct xxstr *xxp = (struct xxstr *)arg;

  xxp->xx_timeoutid = 0;      /* timeout has run */
  /* write your code here */

}
```

Sample Multithreaded Module With Outer Perimeter

Example 12-2 is a sample multithreaded, loadable STREAMS module. The module MT design is a relatively simple one, based on a per queue-pair inner perimeter plus an outer perimeter. The inner perimeter protects per-instance data structure (accessed through the q_ptr field) and the module global data is protected by the outer perimeter. The outer perimeter is configured so that the open and close routines have exclusive access to the outer perimeter. This is necessary because they both modify the global-linked list of instances. Other routines that modify global data are run as qwriter(9F) callbacks, giving them exclusive access to the whole module.

EXAMPLE 12-2   Multithread Module with Outer Perimeter

```
/*
 * Example SunOS 5 multi-threaded STREAMS module.
 * Using a per-queue-pair inner perimeter plus an outer perimeter.
 */

#include  <sys/types.h>
#include  <sys/errno.h>
#include  <sys/stropts.h>
#include  <sys/stream.h>
#include  <sys/strlog.h>
#include  <sys/cmn_err.h>
#include  <sys/kmem.h>
#include  <sys/conf.h>
#include  <sys/ksynch.h>
#include  <sys/modctl.h>
#include  <sys/stat.h>
#include  <sys/ddi.h>
```

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

/*
 * Function prototypes.
 */
static   int xxopen(queue_t *, dev_t *, int, int, cred_t *);
static   int xxclose(queue_t *, int, cred_t *);
static   int xxwput(queue_t *, mblk_t *);
static   int xxwsrv(queue_t *);
static   void xxwput_ioctl(queue_t *, mblk_t *);
static   int xxrput(queue_t *, mblk_t *);
static   void xxtick(caddr_t);

/*
 * Streams Declarations
 */
static struct module_info xxm_info = {
   99,      /* mi_idnum */
   "xx",    /* mi_idname */
   0,       /* mi_minpsz */
   INFPSZ,  /* mi_maxpsz */
   0,       /* mi_hiwat */
   0        /* mi_lowat */
};
/*
 * Define the read-side qinit structure
 */
static struct qinit xxrinit = {
  xxrput,        /* qi_putp */
  NULL,          /* qi_srvp */
  xxopen,        /* qi_qopen */
  xxclose,       /* qi_qclose */
  NULL,          /* qi_qadmin */
  &xxm_info,     /* qi_minfo */
  NULL           /* qi_mstat */
};
/*
 * Define the write-side qinit structure
 */
static struct qinit xxwinit = {
  xxwput,        /* qi_putp */
  xxwsr,         /* qi_srvp */
  NULL,          /* qi_qopen */
  NULL,          /* qi_qclose */
  NULL,          /* qi_qadmin */
  &xxm_info,     /* qi_minfo */
  NULL           /* qi_mstat */
};

static struct streamtab xxstrtab = {
  &xxrini,       /* st_rdinit */
  &xxwini,       /* st_wrinit */
  NULL,          /* st_muxrinit */
  NULL           /* st_muxwrinit */
};

/*
 * define the fmodsw structure.
```

```
 */

static struct fmodsw xx_fsw = {
  "xx",          /* f_name */
  &xxstrtab,     /* f_str */
  (D_NEW|D_MP|D_MTQPAIR|D_MTOUTPERIM|D_MTOCEXCL) /* f_flag */
};

/*
 * Module linkage information for the kernel.
 */
static struct modlstrmod modlstrmod = {
  &mod_strmodops, /* Type of module; a STREAMS module */
  "xx module",  /* Module name */
  &xx_fsw,    /* fmodsw */
};

static struct modlinkage modlinkage = {
  MODREV_1,
  &modlstrmod,
  NULL
};

/*
 * Module private data structure. One is allocated per stream.
 */
struct xxstr {
  struct  xxstr *xx_next; /* pointer to next in list */
  queue_t *xx_rq;    /* read side queue pointer */
  int   xx_timeoutid;  /* id returned from timeout() */
};

/*
 * Linked list of opened stream xxstr structures and other module
 * global data. Protected by the outer perimeter.
 */
static struct xxstr      *xxup = NULL;
static int some_module_global_data;


/*
 * Module Config entry points
 */
int
_init(void)
{
  return (mod_install(&modlinkage));
}
int
_fini(void)
{
  return (mod_remove(&modlinkage));
}
int
_info(struct modinfo *modinfop)
{
  return (mod_info(&modlinkage, modinfop));
}
```

```
static int
xxopen(queue_t *rq,dev_t *devp,int flag,int sflag, cred_t *credp)
{
  struct xxstr *xxp;
  /* If this stream already open - we're done. */
  if (rq->q_ptr)
   return (0);
  /* We must be a module */
  if (sflag != MODOPEN)
   return (EINVAL);

  /*
   * The perimeter flag D_MTOCEXCL implies that the open and
   * close routines have exclusive access to the module global
   * data structures.
   *
   * Allocate our private per-stream data structure.
   */
   xxp = kmem_alloc(sizeof (struct xxstr),KM_SLEEP);

  /* Point q_ptr at it. */
  rq->q_ptr = WR(rq)->q_ptr = (char *) xxp;

  /* Initialize it. */
  xxp->xx_rq = rq;
  xxp->xx_timeoutid = 0;

  /* Link new entry into the list of active entries. */
  xxp->xx_next = xxup;
  xxup = xxp;

  /* Enable xxput() and xxsrv() procedures on this queue. */
  qprocson(rq);
  /* Return success */
  return (0);
}

static int
xxclose(queue_t,*rq, int flag,cred_t *credp)
{
  struct   xxstr    *xxp;
  struct   xxstr    **prevxxp;

  /* Disable xxput() and xxsrv() procedures on this queue. */
  qprocsoff(rq);
  /* Cancel any pending timeout. */
   xxp = (struct xxstr *) rq->q_ptr;
   if (xxp->xx_timeoutid != 0) {
    (void) quntimeout(WR(rq), xxp->xx_timeoutid);
     xxp->xx_timeoutid = 0;
   }
  /*
   * D_MTOCEXCL implies that the open and close routines have
   * exclusive access to the module global data structures.
   *
   * Unlink per-stream entry from the active list and free it.
```

```
    */
  for (prevxxp = &xxup; (xxp = *prevxxp) != NULL;
    prevxxp = &xxp->xx_next) {
   if (xxp == (struct xxstr *) rq->q_ptr)
    break;
  }
  *prevxxp = xxp->xx_next;
  kmem_free (xxp, sizeof (struct xxstr));
  rq->q_ptr = WR(rq)->q_ptr = NULL;
  return (0);
}

static int
xxrput(queue_t, *wq, mblk_t *mp)
{
  struct xxstr *xxp = (struct xxstr *)wq->q_ptr;

  /*
   * Write your code here. Can read "some_module_global_data"
   * since we have shared access at the outer perimeter.
   */
  putnext(wq, mp);
}

/* qwriter callback function for handling M_IOCTL messages */
static void
xxwput_ioctl(queue_t, *wq, mblk_t *mp)
{
  struct xxstr    *xxp = (struct xxstr *)wq->q_ptr;

  /*
   * Write your code here. Can modify "some_module_global_data"
   * since we have exclusive access at the outer perimeter.
   */
  mp->b_datap->db_type = M_IOCNAK;
  qreply(wq, mp);
}

static
xxwput(queue_t *wq, mblk_t *mp)
{
  struct xxstr    *xxp = (struct xxstr *)wq->q_ptr;

  if (mp->b_datap->db_type == M_IOCTL) {
   /* M_IOCTL will modify the module global data */
   qwriter(wq, mp, xxwput_ioctl, PERIM_OUTER);
   return;
  }
  /*
   * Write your code here. Can read "some_module_global_data"
   * since we have exclusive access at the outer perimeter.
   */
  putnext(wq, mp);
}

static
xxwsrv(queue_t wq)
{
```

```
  mblk_t   *mp;
  struct xxstr *xxp= (struct xxstr *) wq->q_ptr;

  while (mp = getq(wq)) {
  /*
   * Write your code here. Can read "some_module_global_data"
   * since we have exclusive access at the outer perimeter.
   */
   freemsg(mp);

   /* for example, start a timeout */
   if (xxp->xx_timeoutid != 0) {
    /* cancel running timeout */
    (void) quntimeout(wq, xxp->xx_timeoutid);
   }
   xxp->xx_timeoutid = qtimeout(wq, xxtick, (char *)xxp, 10);
  }
}

static void
xxtick(arg)
  caddr_t arg;
{
  struct xxstr *xxp = (struct xxstr *)arg;

  xxp->xx_timeoutid = 0;      /* timeout has run */
  /*
   * Write your code here. Can read "some_module_global_data"
   * since we have shared access at the outer perimeter.
   */
}
```

13

STREAMS Multiplex Drivers

This chapter describes how STREAMS multiplexing configurations are created and also discusses multiplexing drivers. A STREAMS *multiplexer* is a driver with multiple streams connected to it. The primary function of the multiplexing driver is to switch messages among the connected streams. Multiplexer configurations are created from the user level by system calls.

This chapter contains the following information:

- "STREAMS Multiplexers" on page 255
- "Connecting And Disconnecting Lower Streams" on page 261
- "Multiplexer Construction Example" on page 263
- "Multiplexing Driver Example" on page 264
- "Persistent Links" on page 272
- "Design Guidelines" on page 274

STREAMS Multiplexers

STREAMS-related system calls are used to set up the "plumbing," or stream interconnections, for multiplexing drivers. The subset of these calls that allows a user to connect (and disconnect) streams below a driver is referred to as *multiplexing*. This type of connection is referred to as a one-to-M, or lower, multiplexer configuration. This configuration must always contain a multiplexing driver, which is recognized by STREAMS as having special characteristics.

Multiple streams can be connected above a driver by open(2) calls. This accommodates the loop-around driver and the driver that handled multiple minor devices in Chapter 9, "STREAMS Drivers". There is no difference between the connections to these drivers. Only the functions performed by the driver are different. In the multiplexing case, the driver routes data between multiple streams. In the device driver case, the driver routes data between user processes and associated physical ports. Multiplexing with streams connected above is referred to as an N-to-1, or upper, multiplexer. STREAMS does not provide any facilities beyond open and close to connect or disconnect upper streams for multiplexing.

From the driver's perspective, upper and lower configurations differ only in the way they are initially connected to the driver. The implementation requirements are the same: route the data

and handle flow control. All multiplexer drivers require special developer-provided software to perform the multiplexing data routing and to handle flow control. STREAMS does not directly support flow control among multiplexed streams. M-to-N multiplexing configurations are implemented by using both of these mechanisms in a driver.

As discussed in Chapter 9, "STREAMS Drivers", the multiple streams that represent minor devices are actually distinct streams in which the driver keeps track of each stream attached to it. The STREAMS subsystem does not recognize any relationship between the streams. The same is true for STREAMS multiplexers of any configuration. The multiplexed streams are distinct and the driver must be implemented to do most of the work.

In addition to upper and lower multiplexers, more complex configurations can be created by connecting streams containing multiplexers to other multiplexer drivers. With such a diversity of needs for multiplexers, providing general-purpose multiplexer drivers is not possible. Rather, STREAMS provides a general purpose multiplexing facility. The facility enables you to set up the intermodule or driver plumbing to create multiplexer configurations of generally unlimited interconnection.

Building a Multiplexer

The example in this section builds a protocol multiplexer with the multiplexing configuration shown in Figure 13-1. To free users from the need to know about the underlying protocol structure, a user-level daemon process is built to maintain the multiplexing configuration. Users can then access the transport protocol directly by opening the transport protocol (TP) driver device node.

An internetworking protocol driver (IP) routes data from a single upper stream to one of two lower streams. This driver supports two STREAMS connections beneath it. These connections are to two distinct networks; one for the IEEE 802.3 standard through the 802.3 driver, and another to the IEEE 802.4 standard through the 802.4 driver. The TP driver multiplexes upper streams over a single stream to the IP driver.

FIGURE   13-1   Protocol Multiplexer
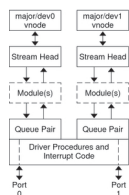
Example 13-1 shows how this daemon process sets up the protocol multiplexer. The necessary declarations and initialization for the daemon program follow.

EXAMPLE 13-1 Protocol Daemon

```
#include <fcntl.h>
#include <stropts.h>
void
main()
{
  int fd_802_4,
     fd_802_3,
     fd_ip,
     fd_tp;
  /* daemon-ize this process */

  switch (fork()) {
   case 0:
    break;
   case -1:
    perror("fork failed");
    exit(2);
   default:
    exit(0);
 }
 (void)setsid();
```

This multilevel multiplexed stream configuration is built from the bottom up. The example begins by first constructing the IP multiplexer. This multiplexing device driver is treated like any other software driver. It owns a node in the file system and is opened just like any other STREAMS device driver.

The first step is to open the multiplexing driver and the 802.4 driver, thus creating separate streams above each driver as shown in Figure 13-2. The stream to the 802.4 driver may now be connected below the multiplexing IP driver using the I_LINK ioctl(2).

FIGURE 13-2 Streams Before Link
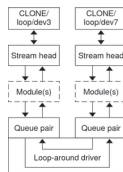
The sequence of instructions to this point is:

```
 if ((fd_802_4 = open("/dev/802_4", O_RDWR)) < 0) {
   perror("open of /dev/802_4 failed");
   exit(1);
 }
 if ((fd_ip = open("/dev/ip", O_RDWR)) < 0) {
   perror("open of /dev/ip failed");
```

```
 exit(2);
}
/* now link 802.4 to underside of IP */
if (ioctl(fd_ip, I_LINK, fd_802_4) < 0) {
 perror("I_LINK ioctl failed");
 exit(3);
}
```

`I_LINK` takes two file descriptors as arguments. The first file descriptor, `fd_ip`, is the stream connected to the multiplexing driver, and the second file descriptor, `fd_802_4`, is the stream to be connected below the multiplexer. The complete stream to the 802.4 driver is connected below the IP driver. The stream head's queues of the 802.4 driver are used by the IP driver to manage the lower half of the multiplexer.

`I_LINK` returns an integer value, `muxid`, which is used by the multiplexing driver to identify the stream just connected below it. `muxid` is ignored in the example, but it is useful for dismantling a multiplexer or routing data through the multiplexer. Its significance is discussed in "Dismantling a Multiplexer" on page 259.

The following sequence of system calls continues building the Internetworking Protocol multiplexer (IP):

```
if ((fd_802_3 = open("/dev/802_3", O_RDWR)) < 0) {
  perror("open of /dev/802_3 failed");
  exit(4);
}
if (ioctl(fd_ip, I_LINK, fd_802_3) < 0) {
 perror("I_LINK ioctl failed");
 exit(5);
}
```

The stream above the multiplexing driver used to establish the lower connections is the controlling stream and has special significance when dismantling the multiplexing configuration. This is illustrated in "Dismantling a Multiplexer" on page 259. The stream referenced by `fd_ip` is the controlling stream for the IP multiplexer.

The order in which the streams in the multiplexing configuration are opened is unimportant. If intermediate modules in the stream are necessary between the IP driver and media drivers, these modules must be added to the streams associated with the media drivers (using `I_PUSH`) before the media drivers are attached below the multiplexer.

The number of streams that can be linked to a multiplexer is restricted by the design of the particular multiplexer. The manual page describing each driver describes such restrictions (see *SunOS Reference Manual*, `Intro(7)`). However, only one `I_LINK` operation is allowed for each lower stream; a single stream cannot be linked below two multiplexers simultaneously.

Continuing with the example, the IP driver is now linked below the transport protocol (TP) multiplexing driver. As seen in Figure 13-1, only one link is supported below the transport driver. This link is formed by the following sequence of system calls:

```
if ((fd_tp = open("/dev/tp", O_RDWR)) < 0) {
  perror("open of /dev/tp failed");
```

```
 exit(6);
}
if (ioctl(fd_tp, I_LINK, fd_ip) < 0) {
 perror("I_LINK ioctl failed");
 exit(7);
}
```

Because the controlling stream of the IP multiplexer has been linked below the TP multiplexer, the controlling stream for the new multilevel multiplexer configuration is the stream above the TP multiplexer.

At this point, the file descriptors associated with the lower drivers can be closed without affecting the operation of the multiplexer. If these file descriptors are not closed, all subsequent read(2), write(2), ioctl(2), poll(2), getmsg(2), and putmsg(2) calls issued to them fail. That is because I_LINK associates the stream head of each linked stream with the multiplexer, so the user may not access that stream directly for the duration of the link.

The following sequence of system calls completes the daemon example:

```
close(fd_802_4);
 close(fd_802_3);
 close(fd_ip);
 /* Hold multiplexer open forever or at least til this process
     is terminated by an external UNIX signal */
 pause();
}
```

The transport driver supports several simultaneous streams. These streams are multiplexed over the single stream connected to the IP multiplexer. The mechanism for establishing multiple streams above the transport multiplexer is actually a by-product of the way in which streams are created between a user process and a driver. By opening different minor devices of a STREAMS driver, separate streams will be connected to that driver. The driver must be designed with the intelligence to route data from the single lower stream to the appropriate upper stream.

The daemon process maintains the multiplexed stream configuration through an open stream (the controlling stream) to the transport driver. Meanwhile, other users can access the services of the transport protocol by opening new streams to the transport driver; they are freed from the need for any unnecessary knowledge of the underlying protocol configurations and subnetworks that support the transport service.

Multilevel multiplexing configurations should be assembled from the bottom up. That is because the passing of ioctl(2) through the multiplexer is determined by the nature of the multiplexing driver and cannot generally be relied on.

Dismantling a Multiplexer

Streams connected to a multiplexing driver from above with open(2), can be dismantled by closing each stream with close(2). The mechanism for dismantling streams that have been

linked below a multiplexing driver is less obvious, and is described in "Disconnecting Lower Streams" on page 263.

I_UNLINK ioctl(2) disconnects each multiplexer link below a multiplexing driver individually. This command has the form:

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

where fd is a file descriptor associated with a stream connected to the multiplexing driver from above, and muxid is the identifier that was returned by I_LINK when a driver was linked below the multiplexer. Each lower driver may be disconnected individually in this way, or a special muxid value of MUXID_ALL can be used to disconnect all drivers from the multiplexer simultaneously.

In the multiplexing daemon program, the multiplexer is never explicitly dismantled. That is because all links associated with a multiplexing driver are automatically dismantled when the controlling stream associated with that multiplexer is closed. Because the controlling stream is open to a driver, only the final call of close for that stream will close it. In this case, the daemon is the only process that has opened the controlling stream, so the multiplexing configuration will be dismantled when the daemon exits.

For the automatic dismantling mechanism to work in the multilevel, multiplexed stream configuration, the controlling stream for each multiplexer at each level must be linked under the next higher-level multiplexer. In the example, the controlling stream for the IP driver was linked under the TP driver. This resulted in a single controlling stream for the full, multilevel configuration. Because the multiplexing program relied on closing the controlling stream to dismantle the multiplexed stream configuration instead of using explicit I_UNLINK calls, the muxid values returned by I_LINK could be ignored.

An important side effect of automatic dismantling on the close is that a process cannot build a multiplexing configuration with I_LINK and then exit. exit(2) closes all files associated with the process, including the controlling stream. To keep the configuration intact, the process must exist for the life of that multiplexer. That is the motivation for implementing the multiplexer as a daemon processs, see "Multiplexing Driver Example" on page 264.

If the process uses persistent links through I_PLINK ioctl(2), the multiplexer configuration remains intact after the process exits. These links are described in "Persistent Links" on page 272.

Routing Data Through a Multiplexer

As demonstrated, STREAMS provides a mechanism for building multiplexed stream configurations. However, the criteria by which a multiplexer routes data are driver dependent. For example, the protocol multiplexer might use address information found in a protocol header to determine the subnetwork over which data should be routed. You must define its routing criteria.

One routing option available to the multiplexer is to use the `muxid` value to determine the stream to which data is routed (remember that each multiplexer link has a `muxid`). `I_LINK` passes the `muxid` value to the driver and returns this value to the user. The driver can therefore specify that the `muxid` value accompany data routed through it. For example, if a multiplexer routed data from a single upper stream to one of several lower streams (as did the IP driver), the multiplexer can require the user to insert the `muxid` of the desired lower stream into the first four bytes of each message passed to it. The driver can then match the `muxid` in each message with the `muxid` of each lower stream, and route the data accordingly.

Connecting And Disconnecting Lower Streams

Multiple streams are created above a driver/multiplexer by use of the open system call on either different minor device, or on a cloneable device file. Note that any driver that handles more than one minor device is considered an upper multiplexer.

To connect streams below a multiplexer requires additional software in the multiplexer. The main difference between STREAMS lower multiplexers and STREAMS device drivers is that multiplexers are pseudo-devices and multiplexers have two additional `qinit` structures, pointed to by fields in `streamtab(9S)`: the lower half read-side `qinit(9S)` and the lower half write-side `qinit(9S)`.

The multiplexer is divided into two parts: the lower half and the upper half. The multiplexer queue structures allocated when the multiplexer was opened use the usual `qinit` entries from the multiplexer's `streamtab(9S)`. This is the same as any open of the STREAMS device. When a lower stream is linked beneath the multiplexer, the `qinit` structures at the stream head are substituted by the lower half `qinit(9S)` structures identified in the `streamstab` for the multiplexers. Once the linkage is made, the multiplexer switches messages between upper and lower streams. When messages reach the top of the lower stream, they are handled by `put` and `service` routines specified in the bottom half of the multiplexer.

Connecting Lower Streams

A lower multiplexer is connected as follows: the initial `open` to a multiplexing driver creates a stream, as in any other driver. `open` uses the `st_rdinit` and `st_wrinit` elements of the `streamtab` structure to initialize the driver queues.. At this point, the only distinguishing characteristics of this stream are non-NULL entries in the `streamtab(9S)` `st_muxrinit` and `st_muxwinit` fields.

These fields are ignored by `open`. Any other stream subsequently opened to this driver will have the same `streamtab` and thereby the same `mux` fields.

Next, another file is opened to create a (soon-to-be) lower stream. The driver for the lower stream is typically a device driver This stream has no distinguishing characteristics. It can include any driver compatible with the multiplexer. Any modules required on the lower stream must be pushed onto it now.

Next, this lower stream is connected below the multiplexing driver with an `I_LINK` ioctl(2) (see streamio(7I)). The stream head points to the stream head routines as its procedures (through its queue). An `I_LINK` to the upper stream, referencing the lower stream, causes STREAMS to modify the contents of the stream head's queues in the lower stream. The pointers to the stream head routines, and other values, in the stream head's queues are replaced with those contained in the `mux` fields of the multiplexing driver's `streamtab`. Changing the stream head routines on the lower stream means that all subsequent messages sent upstream by the lower stream's driver are passed to the `put` procedure designated in `st_muxrinit`, the multiplexing driver. The `I_LINK` also establishes this upper stream as the control stream for this lower stream. STREAMS remembers the relationship between these two streams until the upper stream is closed or the lower stream is unlinked.

Finally, the stream head 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 a linkblk(9S) structure. The multiplexing driver stores information from the linkblk(9S) structure in private storage and returns an `M_IOCACK` acknowledgement. `l_index` is returned to the process requesting the `I_LINK`. This value is used later by the process to disconnect the stream.

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. Any message type can be sent from a lower stream to user processes along any of the upper streams. The upper streams provide the only interface between the user processes and the multiplexer.

No direct data structure linkage is established for the linked streams. The read queue's `q_next` is NULL and the write queue's `q_next` points to the first entity on the lower stream. Messages flowing upstream from a lower driver (a device driver or another multiplexer) will enter the multiplexing driver `put` procedure with the queue represented in `l_qbot` as the `queue_t` for the `put` procedure. The multiplexing driver has to route the messages to the appropriate upper (or lower) stream. Similarly, a message coming downstream from user space on any upper stream has to be processed and routed, if required, by the driver.

Note - It is the responsibility of the driver to handle routing of messages between the upper and lower streams, or between any lateral stream that is part of the multiplexer. This operation is not handled by the STREAMS framework.

In general, multiplexing drivers should be implemented so that new streams can be dynamically connected to (and existing streams disconnected from) the driver without interfering with its ongoing operation. The number of streams that can be connected to a multiplexer is implementation dependent.

Disconnecting Lower Streams

Dismantling a lower multiplexer is accomplished by disconnecting (unlinking) the lower streams. Unlinking can be initiated in three ways:

- An I_UNLINK ioctl(2) referencing a specific stream
- An I_UNLINK indicating all lower streams
- The last close of the control stream

As in the link, an unlink sends a linkblk(9S) structure to the driver in an M_IOCTL message. The I_UNLINK call, which unlinks a single stream, uses the l_index value returned in the I_LINK to specify the lower stream to be unlinked. The latter two calls must designate a file corresponding to a control stream, which causes all the lower streams that were previously linked by this control stream to be unlinked. However, the driver sees a series of individual unlinks.

If no open references exist for a lower stream, a subsequent unlink will automatically close the stream. Otherwise, the lower stream must be closed by close(2) following the unlink. STREAMS automatically dismantles all cascaded multiplexers (below other multiplexing streams) if their controlling stream is closed. An I_UNLINK leaves lower, cascaded multiplexing streams intact unless the stream file descriptor was previously closed.

Multiplexer Construction Example

This section describes an example of multiplexer construction and usage. Multiple upper and lower streams interface to the multiplexer driver.

The Ethernet, LAPB, and IEEE 802.2 device drivers terminate links to other nodes. The multiplexer driver is an Internet Protocol (IP) multiplexer that switches data among the various nodes or sends data upstream to users in the system. The net modules typically provide a convergence function that matches the multiplexer driver and device driver interface.

Streams A, B, and C are opened by the process, and modules are pushed as needed. Two upper streams are opened to the IP multiplexer. The rightmost stream represents multiple streams, each connected to a process using the network. The stream second from the right provides a direct path to the multiplexer for supervisory functions. The control stream, leading to a process, sets up and supervises this configuration. It is always directly connected to the IP driver. Although not shown, modules can be pushed on the control stream.

After the streams are opened, the supervisory process typically transfers routing information to the IP drivers (and any other multiplexers above the IP), and initializes the links. As each link becomes operational, its stream is connected below the IP driver. If a more complex multiplexing configuration is required, the IP multiplexer stream with all its connected links can be connected below another multiplexer driver.

Multiplexing Driver Example

This section contains an example of a multiplexing driver that implements an N-to-1 configuration. This configuration might be used for terminal windows, where each transmission to or from the terminal identifies the window. This resembles a typical device driver, with two differences: the device-handling functions are performed by a separate driver, connected as a lower stream, and the device information (that is, relevant user process) is contained in the input data rather than in an interrupt call.

Each upper stream is created by open(2). A single lower stream is opened and then it is linked by use of the multiplexing facility. This lower stream might connect to the TTY driver. The implementation of this example is a foundation for an M-to-N multiplexer.

As in the loop-around driver (Chapter 9, "STREAMS Drivers"), flow control requires the use of standard and special code because physical connectivity among the streams is broken at the driver. Different approaches are used for flow control on the lower stream, for messages coming upstream from the device driver, and on the upper streams, for messages coming downstream from the user processes.

Note - The code presented here for the multiplexing driver represents a single-threaded, uniprocessor implementation. See Chapter 12, "Multithreaded STREAMS" for details on multiprocessor and multithreading issues such as locking for data corruption and to prevent race conditions.

Example 13-2 is of multiplexer declarations:

EXAMPLE 13-2   Multiplexer Declarations

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/errno.h>
#include <sys/cred.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

static int muxopen (queue_t*, dev_t*, int, int, cred_t*);
static int muxclose (queue_t*, int, cred_t*);
static int muxuwput (queue_t*, mblk_t*);
static int muxlwsrv (queue_t*);
static int muxlrput (queue_t*, mblk_t*);
static int muxuwsrv (queue_t*);

static struct module_info info = {
 0xaabb, "mux", 0, INFPSZ, 512, 128 };

static struct qinit urinit = {       /* upper read */
```

```
 NULL, NULL, muxopen, muxclose, NULL, &info, NULL };

static struct qinit uwinit = {        /* upper write */
 muxuwput, muxuwsrv, NULL, NULL, NULL, &info, NULL };

static struct qinit lrinit = { /* lower read */
 muxlrput, NULL, NULL, NULL, NULL, &info, NULL };

static struct qinit lwinit = { /* lower write */
 NULL, muxlwsrv, NULL, NULL, NULL, &info, NULL };

struct streamtab muxinfo = {
 &urinit, &uwinit, &lrinit, &lwinit };

struct mux {
 queue_t *qptr;     /* back pointer to read queue */
 int  bufcid;     /* bufcall return value */
};
extern struct mux mux_mux[];
extern int mux_cnt;     /* max number of muxes */

static queue_t *muxbot;    /* linked lower queue */
static int muxerr;     /* set if error of hangup on
lower strm */
```

The four `streamtab` entries correspond to the upper read, upper write, lower read, and lower write `qinit` structures. The multiplexing `qinit` structures replace those in each lower stream head (in this case there is only one) after the `I_LINK` has concluded successfully. In a multiplexing configuration, the processing performed by the multiplexing driver can be partitioned between the upper and lower queues. There must be an upper-stream write `put` procedure and lower-stream read `put` procedure. If the queue procedures of the opposite upper/ lower queue are not needed, the queue can be skipped, and the message put to the following queue.

In the example, the upper read-side procedures are not used. The lower-stream read queue `put` procedure transfers the message directly to the read queue upstream from the multiplexer. There is no lower write `put` procedure because the upper write `put` procedure directly feeds the lower write queue downstream from the multiplexer.

The driver uses a private data structure, `mux`. `mux_mux[dev]` points back to the opened upper read queue. This is used to route messages coming upstream from the driver to the appropriate upper queue. It is also used to find a free major or minor device for a `CLONEOPEN` driver open case.

Example 13-3, the upper queue open, contains the canonical driver open code.

EXAMPLE 13-3   Upper Queue Open

```
static int
muxopen(queue_t *q, dev_t *devp, int flag,
     int sflag, cred_t *credp)
{
```

```
  struct mux *mux;
 minor_t device;

 if (q->q_ptr)
   return(EBUSY);

 if (sflag == CLONEOPEN) {
   for (device = 0; device < mux_cnt; device++)
    if (mux_mux[device].qptr == 0)
      break;

   *devp=makedevice(getmajor(*devp), device);
 }
 else {
   device = getminor(*devp);
   if (device >= mux_cnt)
    return ENXIO;
 }

 mux = &mux_mux[device];
 mux->qptr = q;
 q->q_ptr = (char *) mux;
 WR(q)->q_ptr = (char *) mux;
 qprocson(q);
 return (0);
}
```

muxopen checks for a clone or ordinary open call. It initializes q_ptr to point at the mux_mux[] structure.

The core multiplexer processing is as follows: downstream data written to an upper stream is queued on the corresponding upper write message queue if the lower stream is flow controlled. This allows flow control to propagate toward the stream head for each upper stream. A lower write service procedure, rather than a write put procedure, is used so that flow control, coming up from the driver below, may be handled.

On the lower read side, data coming up the lower stream are passed to the lower read put procedure. The procedure routes the data to an upper stream based on the first byte of the message. This byte holds the minor device number of an upper stream. The put procedure handles flow control by testing the upper stream at the first upper read queue beyond the driver.

Upper Write put Procedure Sample

muxuwput, the upper-queue write put procedure, traps ioctl calls, in particular I_LINK and I_UNLINK:

EXAMPLE 13-4   bufcall Callback Routine

```
static int
/*
```

```
* This is our callback routine used by bufcall() to inform us
* when buffers become available
*/
static void mux_qenable(long ql)
{
  queue_t *q = (queue_t *ql);
  struct mux *mux;

  mux = (struct mux *)(q->q_ptr);
  mux->bufcid = 0;
  qenable(q);
}
muxuwput(queue_t *q, mblk_t *mp)
{
 struct mux *mux;

 mux = (struct mux *)q->q_ptr;
 switch (mp->b_datap->db_type) {
 case M_IOCTL: {
   struct iocblk *iocp;
   struct linkblk *linkp;
   /*
    * ioctl. Only channel 0 can do ioctls. Two
    * calls are recognized: LINK, and UNLINK
    */
   if (mux != mux_mux)
    goto iocnak;

   iocp = (struct iocblk *) mp->b_rptr;
   switch (iocp->ioc_cmd) {
   case I_LINK:
    /*
     *Link. The data contains a linkblk structure
     *Remember the bottom queue in muxbot.
     */
    if (muxbot != NULL)
      goto iocnak;

    linkp=(struct linkblk *) mp->b_cont->b_rptr;
    muxbot = linkp->l_qbot;
    muxerr = 0;

    mp->b_datap->db_type = M_IOCACK;
    iocp->ioc_count = 0;
    qreply(q, mp);
    break;
   case I_UNLINK:
    /*
     * Unlink. The data contains a linkblk struct.
     * Should not fail an unlink. Null out muxbot.
     */
    linkp=(struct linkblk *) mp->b_cont->b_rptr;
    muxbot = NULL;
    mp->b_datap->db_type = M_IOCACK;
    iocp->ioc_count = 0;
    qreply(q, mp);
    break;
   default:
```

```
   iocnak:
    /* fail ioctl */
    mp->b_datap->db_type = M_IOCNAK;
    qreply(q, mp);
   }
   break;
}
case M_FLUSH:
   if (*mp->b_rptr & FLUSHW)
    flushq(q, FLUSHDATA);
   if (*mp->b_rptr & FLUSHR) {
    *mp->b_rptr &= ~FLUSHW;
    qreply(q, mp);
   } else
    freemsg(mp);
   break;




case M_DATA:{
    */
    * Data. If we have no lower queue --> fail
    * Otherwise, queue the data and invoke the lower
    * service procedure.
   mblk_t *bp;
    if (muxerr || muxbot == NULL)
     goto bad;
   if ((bp = allocb(1, BPRI_MED)) == NULL) {
     putbq(q, mp);
     mux->bufcid = bufcall(1, BPRI_MED,
      mux_qenable, (long)q);
     break;
    }
    *bp->b_wptr++ = (struct mux *)q->ptr - mux_mux;
    bp->b_cont = mp;
    putq(q, bp);
    break;
}
default:
bad:
   /*
    * Send an error message upstream.
    */
   mp->b_datap->db_type = M_ERROR;
   mp->b_rptr = mp->b_wptr = mp->b_datap->db_base;
   *mp->b_wptr++ = EINVAL;
   qreply(q, mp);
  }
}
```

First, there is a check to enforce that the stream associated with minor device 0 will be
the single, controlling stream. The ioctls are only accepted on this stream. As described
previously, a controlling stream is the one that issues the I_LINK. There should be only a single

control stream. `I_LINK` and `I_UNLINK` include a `linkblk` structure containing the following fields:

`l_qtop` is the upper write queue from which the `ioctl`(2) comes. It always equals `q` for an `I_LINK`, and `NULL` for `I_PLINK`.

`l_qbot` is the new lower write queue. It is the former stream head write queue and is where the multiplexer gets and puts its data.

`l_index` is a unique (system-wide) identifier for the link. It can be used for routing or during selective unlinks. Since the example only supports a single link, `l_index` is not used.

For `I_LINK`, `l_qbot` is saved in `muxbot` and a positive acknowledgement is generated. From this point on, until an `I_UNLINK` occurs, data from upper queues will be routed through `muxbot`. Note that when an `I_LINK`, is received, the lower stream has already been connected. This enables the driver to send messages downstream to perform any initialization functions. Returning an `M_IOCNAK` message (negative acknowledgement) in response to an `I_LINK` causes the lower stream to be disconnected.

The `I_UNLINK` handling code nulls out `muxbot` and generates a positive acknowledgement. A negative acknowledgement should not be returned to an `I_UNLINK`. The stream head ensures that the lower stream is connected to a multiplexer before sending an `I_UNLINK M_IOCTL`.

Drivers can handle the persistent link requests `I_PLINK` and `I_PUNLINK` `ioctl`(2) in the same manner, except that `l_qtop` in the `linkblk` structure passed to the `put` routine is NULL instead of identifying the controlling stream.

`muxuwput` handles `M_FLUSH` messages as a normal driver does, except that there are no messages queued on the upper read queue, so there is no need to call `flushq` if `FLUSHR` is set.

`M_DATA` messages are not placed on the lower write message queue. They are queued on the upper write message queue. When flow control subsides on the lower stream, the lower service procedure, `muxlwsrv`, is scheduled to start output. This is similar to starting output on a device driver.

Upper Write `service` Procedure Sample

The following example shows the code for the upper multiplexer write `service` procedure:

EXAMPLE 13-5   Upper Multiplexer Write Service Procedure

```
static int muxuwsrv(queue_t *q)
{
 mblk_t *mp;
 struct mux *muxp;
 muxp = (struct mux *)q->q_ptr;
```

```
 if (!muxbot) {
   flushq(q, FLUSHALL);
   return (0);
 }
 if (muxerr) {
   flushq(q, FLUSHALL);
   return (0);
 }
 while (mp = getq(q)) {
   if (canputnext(muxbot))
    putnext(muxbot, mp);
   else {
    putbq(q, mp);
    return(0);
   }
 }
 return (0);
}
```

As long as there is a stream still linked under the multiplexer and there are no errors, the service procedure will take a message off the queue and send it downstream, if flow control allows.

Lower Write `service` Procedure

muxlwsrv, the lower (linked) queue write service procedure is scheduled as a result of flow control subsiding downstream (it is back-enabled).

```
static int muxlwsrv(queue_t *q)
{
 int i;

 for (i = 0; i < mux_cnt; i++)
   if (mux_mux[i].qptr && mux_mux[i].qptr->q_first)
    qenable(mux_mux[i].qptr);
 return (0);
}
```

muxlwsrv steps through all possible upper queues. If a queue is active and there are messages on the queue, then its upper write service procedure is enabled through qenable.

Lower Read `put` Procedure

The lower (linked) queue read put procedure is shown in the following example:

EXAMPLE 13-6 Lower Read put Procedure

```
static int
muxlrput(queue_t *q, mblk_t *mp)
```

```
{
 queue_t *uq;
 int device;

 if(muxerr) {
   freemsg(mp);
   return (0);
 }

 switch(mp->b_datap->db_type) {
 case M_FLUSH:
   /*
    * Flush queues. NOTE: sense of tests is reversed
    * since we are acting like a "stream head"
    */
   if (*mp->b_rptr & FLUSHW) {
    *mp->b_rptr &= ~FLUSHR;
    qreply(q, mp);
   } else
    freemsg(mp);
   break;
 case M_ERROR:
 case M_HANGUP:
   muxerr = 1;
   freemsg(mp);
   break;
 case M_DATA:
   /*
    * Route message. First byte indicates
    * device to send to. No flow control.
    *
    * Extract and delete device number. If the
    * leading block is now empty and more blocks
    * follow, strip the leading block.
    */
   device = *mp->b_rptr++;

   /* Sanity check. Device must be in range */
   if (device < 0 || device >= mux_cnt) {
    freemsg(mp);
    break;
   }
   /*
    * If upper stream is open and not backed up,
    * send the message there, otherwise discard it.
    */
   uq = mux_mux[device].qptr;
   if (uq != NULL && canputnext(uq))
    putnext(uq, mp);
   else
    freemsg(mp);
   break;
 default:
   freemsg(mp);
 }
 return (0);
}
```

muxlrput receives messages from the linked stream. In this case, it is acting as a stream head and handles M_FLUSH messages. The code is the reverse of a driver, handling M_FLUSH messages from upstream. There is no need to flush the read queue because no data is ever placed in it.

muxlrput also handles M_ERROR and M_HANGUP messages. If one is received, it locks up the upper streams by setting muxerr.

M_DATA messages are routed by checking the first data byte of the message. This byte contains the minor device of the upper stream. Several checks examine whether:

- The device is in range
- The upper stream is open
- The upper stream is full

This multiplexer does not support flow control on the read side; it is merely a router. If the message passes all checks, it is put to the proper upper queue. Otherwise, the message is discarded.

The upper stream close routine clears the mux entry so this queue will no longer be found. Outstanding bufcalls are not cleared.

```
/*
* Upper queue close
*/
static int
muxclose(queue_t *q, int flag, cred_t *credp)
{
 struct mux *mux;

 mux = (struct mux *) q->q_ptr;
 qprocsoff(q);
 if (mux->bufcid != 0)
  unbufcall(mux->bufcid);
 mux->bufcid = 0;
 mux->ptr = NULL;
 q->q_ptr = NULL;
 WR(q)->q_ptr = NULL;
 return(0);
}
```

Persistent Links

Keeping a process running merely to hold the multiplexer configuration together is not always desirable, so, "free standing" links below a multiplexer are needed. A persistent link is such a link. It is similar to a STREAMS multiplexer link except that a process is not needed to hold the links together. After the multiplexer has been set up, the process may close all file descriptors and exit, and the multiplexer remains intact.

With `I_LINK` and `I_UNLINK` ioctl(2) the file descriptor associated with the stream above the multiplexer used to set up the lower multiplexer connections must remain open for the duration of the configuration. Closing the file descriptor associated with the controlling stream dismantles the whole multiplexing configuration.

Two ioctl(2)s, `I_PLINK` and `I_PUNLINK`, are used to create and remove persistent links that are associated with the stream above the multiplexer. close(2) and `I_UNLINK` are not able to disconnect the persistent links (see strconf(1) and strchg(1)).

The format of `I_PLINK` is:

```
ioctl(fd0, I_PLINK, fd1)
```

The first file descriptor, `fd0`, must reference the stream connected to the multiplexing driver and the second file descriptor, `fd1`, must reference the stream to be connected below the multiplexer. The persistent link can be created as follows:

```
upper_stream_fd = open("/dev/mux", O_RDWR);
lower_stream_fd = open("/dev/driver", O_RDWR);
muxid = ioctl(upper_stream_fd, I_PLINK, lower_stream_fd);
/*
 * save muxid in a file
 */
exit(0);
```

The persistent link can still exist even if the file descriptor associated with the upper stream to the multiplexing driver is closed. The `I_PLINK` ioctl(2) returns an integer value, `muxid`, that can be used for dismantling the multiplexing configuration. If the process that created the persistent link still exists, it may pass the `muxid` value to some other process to dismantle the link, if the dismantling is desired, or it can leave the `muxid` value in a file so that other processes may find it later.

Several users can open the MUX driver and send data to Driver1 since the persistent link to Driver1 remains intact.

The `I_PUNLINK` ioctl(2) is used to dismantle the persistent link. Its format is:

```
ioctl(fd0, I_PUNLINK, muxid)
```

where `fd0` is the file descriptor associated with stream connected to the multiplexing driver from above. The `muxid` is returned by the `I_PLINK` ioctl(2) for the stream that was connected below the multiplexer. `I_PUNLINK` removes the persistent link between the multiplexer referenced by `fd0` and the stream to the driver designated by the `muxid`. Each of the bottom persistent links can be disconnected individually. An `I_PUNLINK` ioctl(2) with the `muxid` value of `MUXID_ALL` will remove all persistent links below the multiplexing driver referenced by `fd0`.

The following code example shows how to dismantle the previously given configuration:

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

```
/*
 * retrieve muxid from the file
 */
ioctl(fd, I_PUNLINK, muxid);
exit(0);
```

Do not use the `I_PLINK` and `I_PUNLINK` `ioctls` with `I_LINK` and `I_UNLINK`. Any attempt to unlink a regular link with `I_PUNLINK` or to unlink a persistent link with the `I_UNLINK` ioctl(2) causes the `errno` value of `EINVAL` to be returned.

Because multilevel multiplexing configurations are allowed in STREAMS, persistent links could exist below a multiplexer whose stream is connected to the above multiplexer by regular links. Closing the file descriptor associated with the controlling stream will remove the regular link but not the persistent links below it. On the other hand, regular links are allowed to exist below a multiplexer whose stream is connected to the above multiplexer with persistent links. In this case, the regular links will be removed if the persistent link above is removed and no other references to the lower streams exist.

The construction of cycles is not allowed when creating links. A cycle could be constructed by:

1. Creating a persistent link of multiplexer 2 below multiplexer 1
2. Closing the controlling file descriptor associated with the multiplexer 2
3. Reopening the file descriptor again
4. Linking the multiplexer 1 below the multiplexer 2

This is not allowed. The operating system prevents a multiplexer configuration from containing a cycle to ensure that messages cannot be routed infinitely, which would create an infinite loop or overflow the kernel stack.

Design Guidelines

The following are general multiplexer design guidelines:

- The upper half of the multiplexer acts like the end of the upper stream. The lower half of the multiplexer acts like the head of the lower stream. Service procedures are used for flow control.
- Message routing is based on multiplexer-specific criteria.
- When one stream is being fed by many streams, flow control may have to take place. Then all feeding streams on the other end of the multiplexer have to be enabled when the flow control is relieved.
- When one stream is feeding many streams, flow control may also have to take place. Be careful not to starve other streams when one becomes flow controlled.
- Upper and lower multiplexers share the same perimeter type and concurrency level. (See "MT STREAMS Perimeters" on page 230 for information about perimeters.)

PART   III

Advanced Topics

	Describes the tools available for debugging STREAMS-based applications.

♦♦♦ **C H A P T E R 1 4**

14

Debugging STREAMS-based Applications

This chapter describes some of the tools available to assist in debugging STREAMS-based applications. It contains the following information:

- "Kernel Debug Printing" on page 277
- "STREAMS Error and Trace Logging" on page 278
- "Kernel Examination Tools" on page 279

Kernel Debug Printing

The kernel routine `cmn_err(9F)` enables printing of formatted strings on a system console. It displays a specified message on the console and can also store it in the `msgbuf` that is a circular array in the kernel. The format of `cmn_err(9F)` is:

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

void cmn_err (int level, char *fmt, int args)
```

where *level* can take the following values:

- `CE_CONT` — may be used as simple `printf(3C)`. It is used to continue another message or to display an informative message not associated with an error.
- `CE_NOTE` — reports system events. It is used to display a message preceded by `NOTICE:`. This message is used to report system events that do not necessarily require user action, but may interest the system administrator. An example is a sector on a disk needing to be accessed repeatedly before it can be accessed correctly.
- `CE_WARN` — reports system events that require user action. This is used to display a message preceded by `WARNING:`. This message is used to report system events that require immediate attention, such as those where, if an action is not taken, the system may panic. For example, when a peripheral device does not initialize correctly, this level should be used.
- `CE_PANIC` — reports system panic. This is used to display a message preceded with `PANIC:`. Drivers should specify this level only under the most severe conditions. A valid use of this level is when the system cannot continue to function. If the error is recoverable

and not essential to continued system operation, do not panic the system. This level halts all processing.

fmt and *args* are passed to the kernel routine `printf` that runs at `splhi` and should be used sparingly. If the first character of `fmt` is `!` (an exclamation point), output is directed to `msgbuf`. `msgbuf` can be accessed with the `crash` command. If the destination character begins with `^` (a caret) output goes to the console. If no destination character is specified, the message is directed to both the `msgbuf` array and the console.

`cmn_err(9F)` appends each `fmt` with "\n", except for the `CE_CONT` level, even when a message is sent to the `msgbuf` array. *args* specifies a set of arguments passed when the message is displayed. Valid specifications are `%s` (string), `%u` (unsigned decimal), `%d` (decimal), `%o` (octal), and `%x` (hexadecimal). `cmn_err(9F)` does not accept length specifications in conversion specifications. For example, `%3d` is ignored.

Note - `crash` has reached EOL; for information about how to transition from `crash` to `mdb`, see the "Oracle Solaris Modular Debugger Guide ".

STREAMS Error and Trace Logging

STREAMS error and trace loggers are provided for debugging and for administering STREAMS modules and drivers. This facility consists of `log(7D)`, `strace(1M)`, `strclean(1M)`, `strerr(1M)`, and `strlog(9F)`.

Any module or driver in any stream can call the STREAMS logging function `strlog(9F)` (see also `log(7D)`). `strlog(9F)` sends formatted text to the error logger `strerr(1M)`, the trace logger `strace(1M)`, or the console logger.

`strerr(1M)` runs as a daemon process initiated at system startup. A call to `strlog(9F)` requesting an error to be logged causes an `M_PROTO` message to be sent to `strerr(1M)`, which formats the contents and places them in a daily file. `strclean(1M)` purges daily log files that have not been modified for three days.

`strlog(9F)` also sends a similar `M_PROTO` message to `strace(1M)`, which places it in a user-designated file. `strace(1M)` is initiated by a user. The user designates the modules and drivers and the severity level of the messages accepted for logging by `strace(1M)`.

A user process can submit its own `M_PROTO` messages to the log driver for inclusion in the logger of its choice through `putmsg(2)`. The messages must be in the same format required by the logging processes and are switched to the loggers requested in the message.

The output to the log files is formatted ASCII text. The files can be processed by standard system commands such as `grep(1)` or by developer-provided routines.

Kernel Examination Tools

Use `crash`, `adb`, and `kadb(1M)` to examine the kernel.

crash Command

`crash` examines kernel structures interactively. It can be used on a system dump and on an active system.

Note - `crash` has reached EOL; for information about how to transition from `crash` to `mdb`, see the *Oracle Solaris Modular Debugger Guide*.

The following `crash` functions are related to STREAMS:

- `dbfree` — Print data block header free list.
- `dblock` — Print allocated STREAMS data block headers.
- `linkblk` — Print the `linkblk(9S)` table.
- `mbfree` — Print free STREAMS message block headers.
- `mblock` — Print allocated STREAMS message block headers.
- `pty` — Print pseudo-TTYs now configured. The `l` option gives information on the line discipline module. `ldterm(7M)`, the `h` option provides information on the pseudo-TTY emulation module `ptem(7M)`. The `s` option gives information on the packet module `pckt(7M)`.
- `qrun` — Print a list of scheduled queues.
- `queue` — Print the STREAMS queues.
- `stream` — Print the `stdata` table.
- `strstat` — Print STREAMS statistics.
- `tty` — Print the `tty` table. The `l` option prints details about the line-discipline module.

  The `crash` functions `dblock`, `linkblk`, `mblock`, `queue`, and `stream` take an optional table entry argument or address that is the address of the data structure. The `strstat` command gives information about STREAMS event cells and `linkblks` in addition to message blocks, data blocks, queues, and streams. On the output report, the `CONFIG` column

represents the number of structures currently configured. It may change because resources are allocated as needed.

adb Command

adb is an interactive general-purpose debugger. It can be used to examine files and provides a controlled environment for the execution of programs. It has no support built in for any STREAMS functionality.

Note - adb has reached EOL; for information about how to transition from adb to mdb, see the *Oracle Solaris Modular Debugger Guide*.

kadb Command

kadb(1M) is an interactive debugger with a user interface similar to adb(1), but it runs in the same virtual address space as the program begin debugged. It also has no specific STREAMS support.

PART   IV

Appendixes

Part IV of this manual contains appendixes that summarize message types, kernel utility interfaces, the STREAMS-based terminal subsystem, and frequently asked questions.

Appendix A, "Message Types", MESSAGE Types	Provides a summary of ordinary and high-priority messages.
Appendix B, "Kernel Utility Interface Summary", Kernel Utility Interface Summary	Provides a summary of the kernel utility interfaces.
Appendix C, "STREAMS-Based Terminal Subsystem", STREAMS-Based Terminal Subsystem	Describes how a terminal subsystem is set up and how interrupts are handled.
Appendix D, "STREAMS FAQ", STREAMS FAQ	Provides a collection of frequently asked questions.

♦ ♦ ♦   **A P P E N D I X   A**

Message Types

STREAMS 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 and sending it in any direction on the stream. However, established processing and direction rules should be observed. Stream head processing according to message type is fixed, although certain parameters can be altered.

- "Ordinary Messages" on page 283
- "High-Priority Messages" on page 291

Ordinary Messages

The message types found in `sys/stream.h` are described in this appendix, classified according to their message queueing priority. Ordinary messages are described first, with high-priority messages following. In certain cases, two message types may perform similar functions, differing only in priority. Message construction is described in Chapter 3, "STREAMS Application-Level Mechanisms" . The use of the word module generally implies *module* or *driver*.

Ordinary messages are also called normal or nonpriority messages. Ordinary messages are subject to flow control whereas high-priority messages are not.

M_BREAK

Sent to a driver to request that `BREAK` be transmitted on whatever media the driver is controlling.

The message format is not defined by STREAMS and its use is developer dependent. This message may be considered a special case of an `M_CTL` message. An `M_BREAK` message cannot be generated by a user-level process and is always discarded if passed to the stream head.

M_CTL

Generated by modules that send information to a particular module or type of module. `M_CTL` messages are typically used for intermodule communication, as when adjacent STREAMS protocol modules negotiate the terms of their interface. An `M_CTL` message cannot be generated by a user-level process and is always discarded if passed to the stream head.

M_DATA

Contains ordinary data. Messages allocated by `allocb`(9F) are `M_DATA` type by default. `M_DATA` messages are generally sent bidirectionally on a stream and their contents can be passed between a process and the stream head. In the `getmsg`(2) and `putmsg`(2) system calls, the contents of `M_DATA` message blocks are referred to as the data part. Messages composed of multiple message blocks typically have `M_DATA` as the message type for all message blocks following the first.

M_DELAY

Sent to a media driver to request a real-time delay on output. The data buffer associated with this message is expected to contain an integer to indicate the number of machine cycles of delay desired. `M_DELAY` messages are typically used to prevent transmitted data from exceeding the buffering capacity of slower terminals.

The message format is not defined by STREAMS and its use is developer dependent. Not all media drivers may understand this message. This message may be considered a special case of an `M_CTL` message. An `M_DELAY` message cannot be generated by a user-level process and is always discarded if passed to the stream head.

M_IOCTL

Generated by the stream head in response to `I_STR`, `I_LINK`, `I_UNLINK`, `I_PLINK`, and `I_PUNLINK` ioctls (see `streamio`(7I)). This message is also generated in response to `ioctl` calls that contain a command argument value not defined in `streamio`(7I). When one of these `ioctl`(2) is received from a user process, the stream head uses values supplied in the call and values from the process to create an `M_IOCTL` message containing them, then sends the message downstream. `M_IOCTL` messages perform the general `ioctl`(2) functions of character device drivers.

For an I_STR ioctl(2), the user values are supplied in a structure of the following form, provided as an argument to the ioctl(2) call (see I_STR in streamio(7I)):

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

where ic_cmd is the request (or command) defined by a downstream module or driver, ic_timout is the time the stream head waits for acknowledgement to the M_IOCTL message before timing out, and ic_dp points to an optional data buffer. On input, ic_len contains the length of the data in the buffer that was passed in. On return from the call, it contains the length of the data, if any, being returned to the user in the same buffer.

The M_IOCTL message format is one M_IOCTL message block followed by zero or more M_DATA message blocks. STREAMS constructs an M_IOCTL message block by placing an iocblk(9S) structure, defined in <sys/stream.h>, in its data buffer. The iocblk(9S) structure differs for 64–bit and 32–bit architectures.

```
#if defined(_LP64)
struct iocblk {
 int     ioc_cmd;      /* ioctl command type */
 cred_t  *ioc_cr;      /* full credentials */
 uint    ioc_id;       /* ioctl id */
 uint    ioc_flag;     /* see below */
 size_t  ioc_count;    /* count of bytes in data field */
 int     ioc_rval;     /* return value  */
 int     ioc_error;    /* error code */
};
#else
struct iocblk {
 int     ioc_cmd;      /* ioctl command type */
 cred_t  *ioc_cr;      /* full credentials */
 uint    ioc_id;       /* ioctl id */
 size_t  ioc_count;    /* count of bytes in data field */
 int     ioc_error;    /* error code */
 int     ioc_rval;     /* return value  */
 intt    ioc_fill1;
 uint    ioc_flag;     /* see below */
 int     ioc_filler[2]; /* reserved for future use */
};
#endif  /* _LP64 */
```

For an I_STR ioctl(2), ioc_cmd corresponds to ic_cmd of the strioctl structure. ioc_cr points to a credentials structure defining the user process's permissions (see cred.hfile).. Its contents can be tested to determine whether the user issuing the ioctl(2) call is authorized to do so. For an I_STR ioctl(2), ioc_count is the number of data bytes, if any, contained in the message and corresponds to ic_len.

ioc_id is an identifier generated internally, and is used by the stream head to match each M_IOCTL message sent downstream with response messages sent upstream to the stream head. The response message that completes the stream-head processing for the ioctl(2) is an M_IOCACK (positive acknowledgement) or an M_IOCNAK (negative acknowledgement) message.

For an I_STR ioctl(2), if a user supplies data to be sent downstream, the stream head copies the data (pointed to by ic_dp in the strioctl structure) into M_DATA message blocks and links the blocks to the initial M_IOCTL message block. ioc_count is copied from ic_len. If there are no data, ioc_count is zero.

If the stream head does not recognize the command argument of an ioctl(2), the head creates a transparent M_IOCTL message. The format of a transparent M_IOCTL message is one M_IOCTL message block followed by one M_DATA block. The form of the iocblk structure is the same as above. However, ioc_cmd is set to the value of the command argument in the ioctl(2) and ioc_count is set to TRANSPARENT, defined in <sys/stream.h>. TRANSPARENT distinguishes the case where an I_STR ioctl(2) specifies a value of ioc_cmd equivalent to the command argument of a transparent ioctl(2). The M_DATA block of the message contains the value of the arg parameter in the ioctl(2).

The first module or driver that understands the ioc_cmd request contained in the M_IOCTL acts on it. For an I_STR ioctl(2), this action generally includes an immediate upstream transmission of an M_IOCACK message. For transparent M_IOCTLs, this action generally includes the upstream transmission of an M_COPYIN or M_COPYOUT message.

Intermediate modules that do not recognize a particular request must pass the message on. If a driver does not recognize the request, or the receiving module can not acknowledge it, an M_IOCNAK message must be returned.

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 following M_DATA blocks. If one of these messages reaches the stream head with an identifier that does not match that of the currently outstanding M_IOCTL message, the response message is discarded. A common means of ensuring that the correct identifier is returned is for the replying module to convert the M_IOCTL message into the appropriate response type and set ioc_count to 0 if no data is returned. Then, qreply(9F) is used to send the response to the stream head.

In an M_IOCACK or M_IOCNAK message, ioc_error holds any return error condition set by a downstream module. If this value is non-zero, it is returned to the user in errno. Note that both an M_IOCNAK and an M_IOCACK can return an error. However, only an M_IOCACK can have a return value. For an M_IOCACK, ioc_rval holds any return value set by a responding module. For an M_IOCNAK, ioc_rval is ignored by the stream head.

If a module processing an I_STR ioctl(2) is sending data to a user process, it must use the M_IOCACK message that it constructs such that the M_IOCACK block is linked to one or more

following `M_DATA` blocks containing the user data. The module must set `ioc_count` to the number of data bytes sent. The stream head places the data in the address pointed to by `ic_dp` in the user `I_STR` `strioctl` structure.

A module processing a transparent `ioctl(2)` that is sending data to a user process can use only an `M_COPYOUT` message. For a transparent `ioctl(2)`, no data can be sent to the user process in an `M_IOCACK` message. All data must be sent in a preceding `M_COPYOUT` message. The stream head ignores any data contained in an `M_IOCACK` message (in `M_DATA` blocks) and frees the blocks.

No data can be sent with an `M_IOCNAK` message for any type of `M_IOCTL`. The stream head ignores and frees any `M_DATA` blocks.

The stream head blocks the user process until an `M_IOCACK` or `M_IOCNAK` response to the `M_IOCTL` (same `ioc_id`) is received. For an `M_IOCTL` generated from an `I_STR` `ioctl(2)`, the stream head times out if no response is received in `ic_timout` interval (the user can specify an explicit interval or specify use of the default interval). For `M_IOCTL` messages generated from all other `ioctl(2)`s, the default (infinite) is used.

M_PASSFP

The `M_PASSFP` message passes a file pointer from the stream head at one end of a stream pipe to the stream head at the other end of the same stream pipe.

The message is generated as a result of an `I_SENDFD` `ioctl(2)` (see `streamio(7I)`) issued by a process to the sending stream head. STREAMS places the `M_PASSFP` message directly on the destination stream head's read queue to be retrieved by an `_RECVFD` `ioctl(2)` (see `streamio(7I)`). The message is placed without passing it through the stream (that is, it is not seen by any modules or drivers in the stream). This message should never be present on any queue except the read queue of a stream head. Consequently, modules and drivers do not need to recognize this message, and it can be ignored by module and driver developers.

M_PROTO

The `M_PROTO` message contains control information and associated data. The message format is one or more `M_PROTO` message blocks followed by zero 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.

> **Note -** On the write side, the user can only generate `M_PROTO` messages containing one `M_PROTO` message block.

The `M_PROTO` message block typically contains implementation dependent control information. `M_PROTO` messages are generally sent bidirectionally on a stream, and their contents can be passed between a process and the stream head. The contents of the first message block of an `M_PROTO` message is generally referred to as the control part, and the contents of any following `M_DATA` message blocks are referred to as the data part. In the `getmsg`(2) and `putmsg`(2), the control and data parts are passed separately.

The format of `M_PROTO` and `M_PCPROTO` (generically `PROTO`) messages sent upstream to the stream head allows multiple `PROTO` blocks at the beginning of the message although its use is not recommended. `getmsg`(2)) compacts the blocks into a single control part when passing them to the user process.

M_RSE

This message is 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_SETOPTS

This message is used to alter some characteristics of the stream head. It is generated by any downstream module, and is interpreted by the stream head. The data buffer of the message has the following structure, as defined in `stream.h`:

```
struct stroptions {
 uint     so_flags;         /* options to set */
 shor     so_readopt;       /* read option */
 ushort   so_wroff;         /* write offset */
 ssize_t  so_minpsz;        /* minimum read packet size */
 ssize_t  so_maxpsz;        /* maximum read packet size */
 size_t   so_hiwat;         /* read queue high−water mark */
 size_t   so_lowat;         /* read queue low−water mark */
 unsigned char so_band;     /* band for water marks */
 ushort   so_erropt;        /* error option */
 ssize_t  so_maxblk;        /* maximum message block size */
 ushort   so_copyopt;       /* copy options (see stropts.h) */
};
```

where `so_flags` specifies which options are to be altered, and 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 (see `read(2)`) as specified by the value of `so_readopt` to:

  - **RNORM** — Byte stream
  - **RMSGD** — Message discard
  - **RMSGN** — Message non-discard
  - **RPROTNORM** — Normal protocol
  - **RPROTDAT** — Turn `M_PROTO` and `M_PCPROTO` msgs into `M_DATA` msgs
  - **RPROTDIS** — Discard `M_PROTO` and `M_PCPROTO` blocks in a `msg` and retain any linked `M_DATA` blocks

- **SO_WROFF** — Direct the stream head to insert an offset (unwritten area), specified by `so_wroff` into the first message block of all `M_DATA` messages created as a result of a `write(2)`. The same offset is inserted into the first `M_DATA` message block, if any, of all messages created by a `putmsg` system call. The default offset is zero.

The offset must be less than the maximum message buffer size (system dependent). Under certain circumstances, a write offset may not be inserted. A module or driver must test that `b_rptr` in the `msgb(9S)` structure is greater than `db_base` in the `datab(9S)` structure to determine that an offset has been inserted in the first message block.

- **SO_MINPSZ** — Change the minimum packet size value associated with the stream head read queue to `so_minpsz`. This value is advisory for the module immediately below the stream head. It should 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** — Change the maximum packet size value associated with the stream head read queue to `so_maxpsz`. This value is advisory for the module immediately below the stream head. It should 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 STREAMS allows.

- **SO_HIWAT** — Change the flow control high–water mark (`q_hiwat` in the `queue(9S)` structure, `qb_hiwat` in the `qband(9S)` structure) on the stream-head read queue to the value specified in `so_hiwat`.

- **SO_LOWAT** — Change the flow control low–water mark (`q_lowat` in the `queue(9S)` structure, `qb_lowat` in the `qband(9S)` structure) on the stream-head read queue to the value specified in `so_lowat`.

- **SO_MREADON** — Enable the stream head to generate `M_READ` messages when processing a `read(2)` system call. If both `SO_MREADON` and `SO_MREADOFF` are set in `so_flags`, `SO_MREADOFF` takes precedence.

- `SO_MREADOFF` — Disable the stream head generation of `M_READ` messages when processing a read(2) system call. This is the default. If both `SO_MREADON` and `SO_MREADOFF` are set in `so_flags`, `SO_MREADOFF` takes precedence.

- `SO_NDELON` — Set non-STREAMS TTY semantics for `O_NDELAY`(or `O_NONBLOCK`) processing on read(2) and write(2). If `O_NDELAY`(or `O_NONBLOCK`) is set, a read(2) returns 0 if no data is waiting to be read at the stream head. If `O_NDELAY`(or `O_NONBLOCK`) is clear, a read(2) blocks until data become available at the stream head. (See note below.)

  Regardless of the state of `O_NDELAY` (or `O_NONBLOCK`), a write(2) blocks on flow control and blocks if buffers are not available.

  If both `SO_NDELON` and `SO_NDELOFF` are set in `so_flags`, `SO_NDELOFF` takes precedence.

Note - For conformance with the POSIX standard, new applications should use the `O_NONBLOCK` flag whose behavior is the same as that of `O_NDELAY` unless otherwise noted.

- `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) blocks until data become available at the stream head. (See note above.)

  If `O_NDELAY` (or `O_NONBLOCK`) is set, a write(2) returns -1 and sets `EAGAIN` if flow control is in effect when the call is received. It blocks if buffers are not available. If `O_NDELAY` (or `O_NONBLOCK`) is set, part of the buffer has been written, and a flow control or "buffers not available" condition is encountered, write(2) terminates and returns 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 `so_flags`, `SO_NDELOFF` takes 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.

- `SO_BAND` — Set watermarks in a band. If the `SO_BAND` flag is set with the `SO_HIWAT` or `SO_LOWAT` flag, the `so_band` field contains the priority band number related to the `so_hiwat` and `so_lowat` fields.

  If the `SO_BAND` flag is not set and the `SO_HIWAT` and `SO_LOWAT` flags are on, the normal high-water 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_ISTTY` — Inform the stream head that the stream is acting like a controlling terminal.
- `SO_ISNTTY` — Inform the stream head that the stream is no longer acting like a controlling terminal.

  For `SO_ISTTY`, the stream may 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_TOSTOP` — Stop on background writes to the stream.
- `SO_TONSTOP` — Do not stop on background writes to the stream. `SO_TOSTOP` and `SO_TONSTOP` are used in conjunction with job control.
- `SO_DELIM` — Messages are delimited.
- `SO_NODELIM` — Messages are not delimited.
- `SO_STRHOLD` — Enable `strwrite` message coalescing.

M_SIG

The `M_SIG` message is sent upstream by modules or drivers to post a signal to a process. When the message reaches the front of the stream-head read queue, it evaluates the first data byte of the message as a signal number, defined in `<sys/signal.h>`. (The signal is not generated until it reaches the front of the stream-head read queue.) The associated signal will be sent to processes under the 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(7I)`).
- 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(2)` 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 `ioctls` set the process group field in the stream head so the stream can generate signals even if it is not a controlling TTY.

High-Priority Messages

High-priority messages are particularly suitable for acknowledging service requests when the acknowledgement should be placed ahead of any other messages at the stream head. High-priority messages are not subject to flow control.

M_COPYIN

The M_COPYIN message is generated by a module or driver and sent upstream to request that the stream head perform a copyin(9F) on behalf of the module or driver. It is valid only after receiving an M_IOCTL message and before an M_IOCACK or M_IOCNAK.

The message format is one M_COPYIN message block containing a copyreq(9S) structure, defined in <sys/stream.h>. Different structures are defined for 64–bit and 32–bit architectures.

```
#if defined(_LP64)
struct copyreq {
 int     cq_cmd;                /* ioctl command (from ioc_cmd) */
 cred_t  *cq_cr;                /* full credentials (from ioc_cmd) */
 uint    cq_id;                 /* ioctl id (from ioc_id) */
 uint    cq_flag;               /* see below */
 mblk_t  *cq_private;           /* private state information */
 caddr_t cq_addr;               /* address to copy data to/from */
 size_t  cq_size;               /* number of bytes to copy */
};
#else
struct copyreq {
 int     cq_cmd;                /* ioctl command (from ioc_cmd) */
 cred_t  *cq_cr;                /* full credentials */
 uint    cq_id;                 /* ioctl id (from ioc_id) */
 caddr_t cq_addr;               /* address to copy data to/from */
 size_t  cq_size;               /* number of bytes to copy */
 uint    cq_flag;               /* see below */
 mblk_t  *cq_private;           /* private state information */
 int     cq_filler[4];          /* reserved for future use */
};
#endif   /* _LP64 */
```

The first four members of the structure correspond to those of the iocblk(9S) structure in the M_IOCTL message that allows the same message block to be reused for both structures. The stream head guarantees that the message block allocated for the M_IOCTL message is large enough to contain a copyreq(9S). The cq_addr field contains the user space address from which the data is to be copied. The cq_size field is the number of bytes to copy from user space. The cq_flag field is reserved for future use and should be set to zero.

This message should not be queued by a module or driver unless it intends to process the data for the ioctl(2).

M_COPYOUT

The M_COPYOUT message is generated by a module or driver and sent upstream to request that the stream head perform a copyout(9F) on behalf of the module or driver. It is valid only after receiving an M_IOCTL message and before an M_IOCACK or M_IOCNAK.

The message format is one `M_COPYOUT` message block followed by one or more `M_DATA` blocks. The `M_COPYOUT` message block contains a `copyreq(9S)` as described in the `M_COPYIN` message with the following differences: the `cq_addr` field contains the user space address to which the data is to be copied. The `cq_size` field is the number of bytes to copy to user space.

Data to be copied to user space is contained in the linked `M_DATA` blocks.

This message should not be queued by a module or driver unless it processes the data for the `ioctl` in some way.

Note - For more information, see `copyin` and `copyout` in the "Writing Device Drivers for Oracle Solaris 11.2 " manual.

M_ERROR

The `M_ERROR` message is sent upstream by modules or drivers to report a downstream error condition. When the message reaches the stream head, the stream is marked so that all subsequent system calls issued to the stream, excluding `close(2)` and `poll(2)`, fail with `errno` set to the first data byte of the message. `POLLERR` is set if the stream is being polled. All processes sleeping on a system call to the stream are awakened. An `M_FLUSH` message with `FLUSHRW` is sent downstream.

The stream head maintains two error fields, one for the read side and one for the write side. The one-byte format `M_ERROR` message sets both of these fields to the error specified by the first byte in the message.

There is also a two-byte format of the `M_ERROR` message. The first byte is the read error and the second byte is the write error. This enables modules to set a different error on the read side and write side. If one of the bytes is set to `NOERROR`, then the field for the corresponding side of the stream is unchanged. The module can then ignore an error on one side of the stream. For example, if the stream head was not in an error state and a module sent an `M_ERROR` message upstream with the first byte set to `EPROTO` and the second byte set to `NOERROR`, all subsequent read-like system calls (such as `read(2)`and `getmsg(2)`) fail with `EPROTO`, but all write-like system calls (such as `write(2)` and `putmsg(2)`) still succeed. If a byte is set to `0`, the error state is cleared for the corresponding side of the stream. The values `NOERROR` and `0` are not valid for the one-byte form of the `M_ERROR` message.

M_FLUSH

The M_FLUSH message requests all modules and drivers that receive it to flush their message queues (discard all messages in those queues) as indicated in the message. An M_FLUSH can originate at the stream head, or in any module or driver. The first byte of the message contains flags that specify one of the following actions:

- 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

Each module passes this message to its neighbor after flushing its appropriate queues until the message reaches one of the ends of the stream.

Drivers are expected to include the following processing for M_FLUSH messages. When an M_FLUSH message is sent downstream through the write queues in a stream, the driver at the stream end discards it if the message action indicates that the read queues in the stream are not to be flushed (only FLUSHW set). If the message indicates that the read queues are to be flushed, the driver shuts off the FLUSHW flag, and sends the message up the stream's read queues.

When a flush message is sent up a stream's read side, the stream head checks to see if the write side of the stream is to be flushed. If only FLUSHR is set, the stream head discards the message. However, if the write side of the stream is to be flushed, the stream head sets the M_FLUSH flag to FLUSHW and sends the message down the stream's write side. All modules that queue messages must identify and process this message type.

If FLUSHBAND is set, the second byte of the message contains the value of the priority band to flush.

M_HANGUP

The M_HANGUP message is sent upstream by a driver to report that it can no longer send data upstream. For example, this might be due to an error, or to a remote line connection being dropped. When the message reaches the stream head, the stream is marked so that all subsequent write(2) and putmsg(2) calls issued to the stream will fail and return an ENXIO error. Those ioctls that cause messages to be sent downstream are also failed. POLLHUP is set if the stream is being polled.

Subsequent read(2) or getmsg(2) calls to the stream will not generate an error. These calls will return any messages (according to their function) that were on, or in transit to, the stream-

head read queue before the `M_HANGUP` message was received. When all such messages have been read, `read(2)` returns 0 and `getmsg(2)` will set each of its two length fields to 0.

This message also causes a `SIGHUP` signal to be sent to the controlling process instead of the foreground process group, as the allocation and deallocation of controlling terminals to a session is the responsibility of the controlling process.

M_IOCACK

The `M_IOCACK` message signals the positive acknowledgement of a previous `M_IOCTL` message. The message format is one `M_IOCACK` block (containing an `iocblk(9S)` structure, see `M_IOCTL`) followed by zero or more `M_DATA` blocks. The `iocblk(9S)` may contain a value in `ioc_rval` to be returned to the user process. It may also contain a value in `ioc_error` to be returned to the user process in `errno`.

If this message is responding to an `I_STR` ioctl (see `streamio(7I)`), it may contain data from the receiving module or driver to be sent to the user process. In this case, message format is one `M_IOCACK` block followed by one or more `M_DATA` blocks containing the 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 cannot be returned in an `M_IOCACK` message responding to a transparent `M_IOCTL`. The data must have been sent with preceding `M_COPYOUT` messages. If any `M_DATA` blocks follow the `M_IOCACK` block, the stream head ignores and frees them.

The format and use of this message type is described further under `M_IOCTL`.

M_IOCDATA

The `M_IOCDATA` message is generated by the stream head and sent downstream as a response to an `M_COPYIN` or `M_COPYOUT` message. The message format is one `M_IOCDATA` message block followed by zero or more `M_DATA` blocks. The `M_IOCDATA` message block contains a `copyresp(9S)`, defined in `sys/stream.h`.

```
#if     defined(_LP64)
struct copyresp {
 int     cp_cmd;          /* ioctl command (from ioc_cmd) */
 cred_t  *cp_cr;          /* full credentials (from ioc_cmd) */
 uint    cp_id;           /* ioctl id (from ioc_id) */
 uint    cp_flag;         /* see above */
 mblk_t  *cp_private;     /* private state information */
 caddr_t cp_rval;         /* status of request: 0 -> success */
```

```
                                    /*   non-zero -> failure */
            };
            #else
            struct copyresp {
             int      cp_cmd;          /* ioctl command (from ioc_cmd) */
             cred_t   *cp_cr;          /* full credentials */
             uint     cp_id;           /* ioctl id (from ioc_id) */
             caddr_t  cp_rval          /* status of request: 0 -> success */
                                       /*   non-zero -> failure */
             size_t   cp_pad1;
             uint     cp_pad2;
             mblk_t   *cp_private;     /* private state information */
             uint     cp_flag;         /* see above */
             int      cp_filler[3];
            };
            #endif  /* _LP64 */
```

The first three members of the structure correspond to those of the `iocblk(9S)` in the `M_IOCTL` message that allows the same message blocks to be reused for all of the related transparent messages (`M_COPYIN`, `M_COPYOUT`, `M_IOCACK`, `M_IOCNAK`). The `cp_rval` 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 should not generate an `M_IOCNAK` message. It must abort all `ioctl(2)` processing, clean up its data structures, and return.

The `cp_private` field is copied from the `cq_private` field in the associated `M_COPYIN` or `M_COPYOUT` message. It is included in the `M_IOCDATA` message so the message can be self-describing. This is intended to simplify `ioctl(2)` processing by modules and drivers.

If the message is in response to an `M_COPYIN` message and success is indicated, the `M_IOCDATA` block is followed by `M_DATA` blocks containing the data copied in.

If an `M_IOCDATA` block is reused, any unused fields defined for the resultant message block should be cleared (particularly in an `M_IOCACK` or `M_IOCNAK`).

This message should not be queued by a module or driver unless it processes the data for the `ioctl` in some way.

M_IOCNAK

The `M_IOCNAK` message signals the negative acknowledgement (failure) of a previous `M_IOCTL` message. Its form is one `M_IOCNAK` block containing an `iocblk(9S)`. The `iocblk(9S)` can contain a value in `ioc_error` to be returned to the user process in `errno`. Unlike the `M_IOCACK`, no user data or return value can be sent with this message. If any `M_DATA` blocks follow the `M_IOCNAK` block, the stream head ignores and frees them. When the stream head receives an `M_IOCNAK`, the outstanding `ioctl(2)` request, if any, fails. The format and use of this message type is described further under `M_IOCTL`.

M_PCPROTO

The `M_IOCPROTO` message is the same as the `M_PROTO` message type, except for the priority and the following additional attributes. When an `M_PCPROTO` message is placed on a queue, its service procedure is always enabled. The stream head allows only one `M_PCPROTO` message to be placed in its read queue at a time. If an `M_PCPROTO` message is already in the queue when another arrives, the second message is discarded and its message blocks freed.

This message is intended to allow data and control information to be sent outside the normal flow control constraints.

`getmsg(2)` and `putmsg(2)` refer to messages as high priority messages.

M_PCRSE

The `M_PCRSE` message is 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

The `M_PCSIG` message is the same as the `M_SIG` message, except for the priority. `M_PCSIG` is often preferable to `M_SIG` especially in TTY applications, because `M_SIG` may be queued while `M_PCSIG` is more likely to get through quickly. For example, if an `M_SIG` message is generated when the DEL (delete) key is pressed on the terminal and the user has already typed ahead, the `M_SIG` message becomes queued and the user does not get the call until too late, becoming impossible to kill or interrupt a process by pressing a delete key.

M_READ

The `M_READ` message is generated by the stream head and sent downstream for a `read(2)` if no messages are waiting to be read at the stream head and if read notification has been enabled. Read notification is enabled with the `SO_MREADON` flag of the `M_SETOPTS` message and disabled by use of the `SO_MREADOFF` flag.

The message content is set to the value of the `nbyte` parameter (the number of bytes to be read) in `read(2)`.

`M_READ` notifies modules and drivers of the occurrence of a read. It also supports communication between streams that reside in separate processors. The use of the `M_READ` message is developer

dependent. Modules may take specific action and pass on or free the `M_READ` message. Modules that do not recognize this message must pass it on. All other drivers may or may not take action and then free the message.

This message cannot be generated by a user-level process and should not be generated by a module or driver. It is always discarded if passed to the stream head.

SO_MREADOFF and M_STOP

The `SO_MREADOFF` and `M_STOP` messages request devices to start or stop their output. They are used to produce momentary pauses in a device's output, not to turn devices on or off.

The message format is not defined by STREAMS and its use is developer dependent. These messages may be considered special cases of an `M_CTL` message. These messages cannot be generated by a user-level process and each is always discarded if passed to the stream head.

SO_MREADOFFI and M_STOPI

The `SO_MREADOFFI` and `M_STOPI` messages are the same as `SO_MREADOFF` and `M_STOP` except that `SO_MREADOFFI` and `M_STOPI` are used to start and stop input.

M_UNHANGUP

The `M_UNHANGUP` message reconnects the carrier after it has been dropped.

B

♦ ♦ ♦   **A P P E N D I X   B**

Kernel Utility Interface Summary

Use the following kernel utilities to develop and maintain drivers.

TABLE B-1   Kernel Utility Interfaces

Routine	Description
adjmsg(9F)	Trim bytes in a message
allocb(9F)	Allocate a message block
backq(9F)	Get pointer to the queue behind a given queue
bcanput(9F)	Test for flow control in a given priority band
bufcall(9F)	Recover from failure of allocb allocb(9F)
canput(9F)	Test for room in a queue
copymsg(9F)	Copy a message
datamsg(9F)	Test whether message is a data message
dupb(9F)	Test whether thread can receive user-level signal
dupb(9F)	Duplicate a message block descriptor
dupmsg(9F)	Duplicate a message
enableok(9F)	Re-allow a queue to be scheduled for service
esballoc(9F)	Allocate message and data blocks
flushband(9F)	Flush messages in a given priority band
flushq(9F)	Flush a queue
freeb(9F)	Free a message block
freemsg(9F)	Free all message blocks in a message
freezestr(9F)	Disable changes to the state of the stream

Routine	Description
getq(9F)	Get a message from a queue
insq(9F)	Put a message at a specific place in a queue
linkb(9F)	Concatenate two messages into one
msgdsize(9F)	Get the number of data bytes in a message
noenable(9F)	Prevent a queue from being scheduled
OTHERQ(9F)	Get pointer to the queue's partner queue
pullupmsg(9F)	Concatenate and align bytes in a message
putbq(9F)	Return a message to the beginning of a queue
putctl(9F)	Put a control message
putctl1(9F)	Put a control message with a one-byte parameter
putnext(9F)	Put a message to the next queue
putq(9F)	Put a message on a queue
qbufcall(9F)	Call a function when a buffer becomes available
qprocsoff(9F)	Turn off queue processing
qprocson(9F)	Turn on queue processing
qreply(9F)	Send a message on a stream in the reverse direction
qsize(9F)	Find the number of messages on a queue
qtimeout(9F)	Execute a function after a specified length of time
qunbufcall(9F)	Cancel a pending qbufcall request
quntimeout(9F)	Cancel a pending qtimeout request
qwait(9F)	Perimeter wait routine
qwait_sig(9F)	Perimeter wait routine
qwriter(9F)	Asynchronous perimeter upgrade
RD(9F)	Get pointer to the read queue
rmvb(9F)	Remove a message block from a message
rmvq(9F)	Remove a message from a queue
strlog(9F)	Submit messages for logging

Routine	Description
strqget(9F)	Obtain information on a queue or a band of the queue
strqset(9F)	Change information on a queue or a band of the queue
testb(9F)	Check for an available buffer
testb(9F)	Cancel bufcall request
unfreezestr(9F)	Enable changes to the state of the stream
unlinkb(9F)	Remove a message block from the head of a message
WR(9F)	Get pointer to the write queue

C

♦ ♦ ♦   **A P P E N D I X   C**

STREAMS-Based Terminal Subsystem

This chapter describes how a terminal subsystem is set up and how interrupts are handled. Different protocols are addressed, as well as canonical processing and line discipline substitution.

- "Overview of Terminal Subsystem" on page 303
- "STREAMS-based Pseudo-Terminal Subsystem" on page 312

Overview of Terminal Subsystem

STREAMS provides a uniform interface for implementing character I/O devices and networking protocols in the kernel. The SunOS 5.6 software implements the terminal subsystem in STREAMS. The STREAMS-based terminal subsystem (Figure C-1) provides many benefits:

- **Reusable line discipline modules**. The same module can be used in many streams where the configuration of these streams may be different.
- **Line-discipline substitution**. Although the Oracle Solaris operating environment provides a standard terminal line-discipline module, another one conforming to the interface can be substituted. For example, a remote login feature may use the terminal subsystem line discipline module to provide a terminal interface to the user.
- **Internationalization**. The modularity and flexibility of the STREAMS-based terminal subsystem enables an easy implementation of a system that supports multiple-byte characters for internationalization. This modularity also allows easy addition of new features to the terminal subsystem.
- **Easy customizing**. Users may customize their terminal subsystem environment by adding and removing modules of their choice.
- **The pseudo-terminal subsystem**. The pseudo-terminal subsystem can be easily supported (this is discussed in more detail in the section "STREAMS-based Pseudo-Terminal Subsystem" on page 312).
- **Merge with networking**. By pushing a line discipline module on a network line, you can make the network look like a terminal line.

FIGURE C-1 STREAMS-based Terminal Subsystem
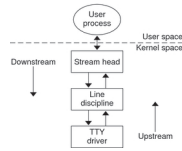
The initial setup of the STREAMS-based terminal subsystem is handled with the ttymon(1M) command within the framework of the Service Access Facility (*autopush* feature). See "STREAMS Administrative Driver" on page 219 for more information.

The STREAMS-based terminal subsystem supports termio(7I), the termios(3C) specification of the POSIX standard, multiple-byte characters for internationalization, the interface to asynchronous hardware flow control and peripheral controllers for asynchronous terminals. XENIX™ and BSD compatibility can also be provided by pushing the ttcompat module.

To use shl with the STREAMS-based terminal subsystem, the sxt driver is implemented as a STREAMS-based driver. The sxt feature is being discontinued and might not be available, so try to use the job-control mechanism instead of sxt. Note that both shl and job control should *not* be run simultaneously.

Master Driver and Slave Driver Characteristics

The master driver and slave driver have the following characteristics:

- Each master driver has one-to-one relationship with a slave device based on major/minor device numbers.
- Only one open is allowed on a master device. Multiple opens are allowed on the slave device according to standard file mode and ownership permissions.
- Each slave driver minor device has a node in the file system.
- An open on a master device automatically locks out an open on the corresponding slave driver.
- A slave cannot be opened unless the corresponding master is open and has unlocked the slave.
- To provide a TTY interface to the user, the ldterm and ptem modules are pushed on the slave side.
- A close on the master sends a hang up to the slave and renders both streams unusable, after all data have been consumed by the process on the slave side.

- The last close on the slave side sends a zero-length message to the master but does not sever the connection between the master and slave drivers.

Line-Discipline Module

A STREAMS line-discipline module called `ldterm` is a key part of the STREAMS-based terminal subsystem. Throughout this chapter, the terms *line discipline* and `ldterm` are used interchangeably and refer to the STREAMS version of the standard line discipline and not the traditional character version. `ldterm` performs the standard terminal I/O processing traditionally done through the `linesw` mechanism.

The `termio`(7I) and `termios`(3C) specifications describe four flags that are used to control the terminal:

- `c_iflag` defines input modes
- `c_oflag` defines output modes
- `c_cflag` defines hardware control modes
- `c_lflag` defines terminal functions used by `ldterm`(7M)

To process these flags elsewhere (for example, in the firmware or in another process), a mechanism is in place to turn the processing of these flags on and off. When `ldterm`(7M) is pushed, it sends an `M_CTL` message downstream that asks the driver which flags the driver will process. The driver sends back that message in response if it needs to change the `ldterm` default processing. By default, `ldterm`(7M) assumes that it must process all flags except `c_cflag` unless it receives a message indicating otherwise.

Default Settings

When `ldterm` is pushed on the stream, the `open` routine initializes the settings of the `termio` flags. The default settings are:

```
c_iflag = BRKINT|ICRNL|IXON|IMAXBEL
c_oflag = OPOST|ONLCR|TAB3
c_cflag = CREAD|CS8|B9600
c_lflag = ISIG|ICANON|ECHO|ECHOK|IEXTEN|ECHOE|ECHOKE | ECHOCTL
```

In canonical mode (`ICANON` flag in `c_lflag` is turned on), `read` from the terminal file descriptor is in message non-discard (`RMSGN`) mode (see `streamio`(7I)). This implies that in canonical mode, `read` on the terminal file descriptor always returns at most one line, regardless of how many characters have been requested. In non-canonical mode, `read` is in byte-stream (`RNORM`) mode. The flag `ECHOCTL` has been added for SunOS 4.1 compatibility.

For information on user-configurable settings, see `termio`(7I).

Module `open` and `close` Routines

The `open` routine of the `ldterm(7M)` module allocates space for holding the TTY structure (see `ldtermstd_state_t` in `ldterm.h`) by allocating a buffer from the STREAMS buffer pool. The number of modules that can be pushed on one stream, as well as the number of TTYs in use, is limited. The number of instances of `ldterm` that have been pushed is limited only by available memory. The `open` also sends an `M_SETOPTS` message upstream to set the stream head high-water and low-water marks to 1024 and 200, respectively. These are the current values.

The `ldterm` module identifies itself as a TTY to the stream head by sending an `M_SETOPTS` message upstream with the `SO_ISTTY` bit of `so_flags` set. The stream head allocates the controlling TTY on the open, if one is not already allocated.

To maintain compatibility with existing applications that use the `O_NDELAY` flag, the `open` routine sets the `SO_NDELON` flag on in the `so_flags` field of the `stroptions(9S)` structure in the `M_SETOPTS` message.

The `open` routine fails if there are no buffers available (so it cannot allocate the internal state structure) or when an interrupt occurs while waiting for a buffer to become available.

The `close` routine frees all the outstanding buffers allocated by this stream. It also sends an `M_SETOPTS` message to the stream head to undo the changes made by the open routine. The `ldterm(7M)` module also sends `M_START` messages downstream to undo the effect of any previous `M_STOP` messages.

Read-Side Processing

The `ldterm(7M)` module's read-side processing has `put` and `service` procedures. High-water and low-water marks for the read queue are 1024 and 200, respectively. These are the current values.

`ldterm(7M)` can send the following messages upstream:

`M_DATA, M_BREAK, M_PCSIG, M_SIG, M_FLUSH, M_ERROR, M_IOCACK, M_IOCNAK, M_HANGUP, M_CTL, M_SETOPTS, M_COPYOUT,` and `M_COPYIN.`

The `ldterm(7M)` module's read side processes `M_BREAK, M_DATA, M_CTL, M_FLUSH, M_HANGUP, M_IOCACK` and `M_IOCNAK` messages. All other messages are sent upstream unchanged.

The `put` procedure scans the message for flow-control characters (`IXON`), signal-generating characters, and, after (possible) transformation of the message, queues the message for the `service` procedure. Echoing is handled completely by the `service` procedure.

If the `ICANON` flag is on in `c_lflag`, canonical processing is performed. If the `ICANON` flag is off, non-canonical processing is performed (see `termio(7I)` for more details). Handling of

VMIN/VTIME in the STREAMS environment is somewhat complicated, because read needs to activate a timer in the ldterm module in some cases; hence, read notification becomes necessary. When a user issues an ioctl(2) to put ldterm(7M) in non-canonical mode, the module sends an M_SETOPTS message to the stream head to register read notification. Further reads on the terminal file descriptor will cause the stream head to issue an M_READ message downstream and data will be sent upstream in response to the M_READ message. With read notification, buffering of raw data is performed by ldterm(7M). Canonizing the raw data when the user has switched from raw to canonical mode is possible. However, the reverse is not possible.

To summarize, in non-canonical mode, the ldterm(7M) module buffers all data until VMIN or VTIME criteria are met. For example, if VMIN=3 and VTIME=0, and three bytes have been buffered, these characters are sent to the stream head *regardless* of whether there is a pending M_READ, and no M_READ needs to be sent downstream. If an M_READ message is received, the number of bytes sent upstream is the argument of the M_READ message unless VTIME is satisfied before VMIN (for example. the timer has expired) , in which case whatever characters are available will be sent upstream.

The service procedure of ldterm(7M) handles STREAMS-related flow control. Because the read side high-water and low-water marks are 1024 and 200 respectively, placing 1024 characters or more on the read queue causes the QFULL flag be turned on, indicating that the module below should not send more data upstream.

Input flow control is regulated by the line-discipline module which generates M_STARTI and M_STOPI high priority messages. When sent downstream, receiving drivers or modules take appropriate action to regulate the sending of data upstream. Output flow control is activated when ldterm(7M) receives flow control characters in its data stream. The module then sets an internal flag indicating that output processing is to be restarted/stopped and sends an M_START/M_STOP message downstream.

Write-Side Processing

Write-side processing of the ldterm(7M) module is performed by the write-side put and service procedures.

The ldterm module supports the following ioctls:

TCSETA, TCSETAW, TCSETAF, TCSETS, TCSETSW, TCSETSF, TCGETA, TCGETS, TCXONC, TCFLSH, and TCSBRK.

All ioctls not recognized by the ldterm(7M) module are passed downstream to the neighboring module or driver.

The following messages can be received on the write side:

M_DATA, M_DELAY, M_BREAK, M_FLUSH, M_STOP, M_START, M_STOP, M_START, M_READ, M_IOCDATA, M_CTL, and M_IOCTL.

On the write side, the ldterm module processes M_FLUSH, M_DATA, M_IOCTL, and M_READ messages, and all other messages are passed downstream unchanged.

An M_CTL message is generated by ldterm(7M) as a query to the driver for an intelligent peripheral and to decide on the functional split for termio(7I) processing. If all or part of termio(7I) processing is done by the intelligent peripheral, ldterm(7M) can turn off this processing to avoid computational overhead. This is done by sending an appropriate response to the M_CTL message, as follows:

- If all of the termio(7I) processing is done by the peripheral hardware, the driver sends an M_CTL message back to ldterm(7M) with ioc_cmd of the structure iocblk(9S) set to MC_NO_CANON. If ldterm(7M) is to handle all termio(7I) processing, the driver sends an M_CTL message with ioc_cmd set to MC_DO_CANON. The default is MC_DO_CANON.

- If the peripheral hardware handles only part of the termio(7I) processing, it informs ldterm(7M) in the following way:

  The driver for the peripheral device allocates an M_DATA message large enough to hold a termios(3C) structure. The driver then turns on those c_iflag, c_oflag, and c_lflag fields of the termios(3C) structure that are processed on the peripheral device by executing an OR operation on the flag values. The M_DATA message is then attached to the b_cont field of the M_CTL message it received. The message is sent back to ldterm(7M) with ioc_cmd in the data buffer of the M_CTL message set to MC_PART_CANON.

One difference between AT&T STREAMS and SunOS 5 STREAMS is that AT&T's line discipline module does not check whether write-side flow control is in effect before forwarding data downstream. It expects the downstream module or driver to add the messages to its queue until flow control is lifted. This is not true in SunOS 5 STREAMS.

EUC Handling in ldterm

Post-processing (the o_flags) should not be handled by the host processor unless the board software is prepared to deal with international (EUC) character sets properly because that post-processing must take the EUC information into account. ldterm(7M) allots the appropriate screen width of characters (that is, how many columns are taken by characters from each given code set on the current physical display) and it takes this width into account when calculating tab expansions. When using multi-byte characters or multi-column characters ldterm automatically handles tab expansion (when TAB3 is set) and does not leave this handling to a lower module or driver.

By default, multi-byte handling by `ldterm` is turned off. When `ldterm` receives an `EUC_WSET` `ioctl(2)`, it turns multi-byte processing on if it is essential to properly handle the indicated code set. Thus, if you use single byte 8-bit codes and have no special multi-column requirements, the special multi-column processing is not used at all. This means that multi-byte processing does not reduce the processing speed or efficiency of `ldterm` unless it is actually used.

The following describes how the EUC handling in `ldterm` works:

First, the multi-byte and multi-column character handling is only enabled when the `EUC_WSET` `ioctl` indicates that one of the following conditions is met:

- Code set consists of more than one byte (including the `SS2` and/or `SS3`) of characters
- Code set requires more than one column to display on the current device, as indicated in the `EUC_WSET` structure

Assuming that one or more of the previous conditions exists, EUC handling is enabled. At this point, a parallel array (see `ldterm_mod` structure) used for other information is allocated and a pointer to it is stored in `t_eucp_mp`. The parallel array that it holds is pointed to by `t_eucp`. The `t_codeset` field holds the flag that indicates which of the code sets is currently being processed on the read side. When a byte with the high bit arrives, `ldterm` checks to see if it is `SS2` or `SS3`. If yes, it belongs to code set 2 or 3. Otherwise, it is a byte that comes from code set 1.

Once the extended code set flag has been set, the input processor retrieves the subsequent bytes, as they arrive, to build one multi-byte character. The counter field `t_eucleft` tells the input processor how many bytes remain to be read for the current character. The parallel array `t_eucp` holds its display width for each logical character in the canonical buffer. During erase processing, positions in the parallel array are consulted to determine how many backspaces need to be send to erase each logical character. (In canonical mode, one backspace of input erases one logical character, no matter how many bytes or columns that character consumes.) This greatly simplifies erase processing for EUC.

The `t_maxeuc` field holds the maximum length, in memory bytes, of the EUC character mapping currently in use. The `eucwioc` field is a substructure that holds information about each extended code set.

The `t_eucign` field aids in output post-processing (tab expansion). When characters are output, `ldterm(7M)` keeps a column to indicate what the current cursor column is supposed to be. When it sends the first byte of an extended character, it adds the number of columns required for that character to the output column. It then subtracts one from the total width in memory bytes of that character and stores the result in `t_eucign`. This field tells `ldterm(7M)` how many subsequent bytes to ignore for the purposes of column calculation. (`ldterm(7M)` calculates the appropriate number of columns when it sees the first byte of the character.)

The field `t_eucwarn` is a counter for occurrences of bad extended characters. It is mostly useful for debugging. After receiving a certain number of illegal EUC characters (perhaps because of some problem on the line or with declared values), a warning is given on the system console.

There are two relevant files for handling multi-byte characters: `euc.h` and `eucioctl.h`. `eucioctl.h` contains the structure that is passed with `EUC_WSET` and `EUC_WGET` calls. The normal way to use this structure is to get `CSWIDTH` from the locale using a mechanism such as `getwidth`(3C) or `setlocale`(3C), copy the values into the structure in `eucioctl.h`, and send the structure using an `I_STR` `ioctl`(2). The `EUC_WSET` call informs the `ldterm`(7M) module about the number of bytes in extended characters and how many columns the extended characters from each set consume on the screen. This enables `ldterm`(7M) to treat multi-byte characters as single units for the purpose of erase processing and to correctly calculate tab expansions for multi-byte characters.

Note - `LC_CTYPE` (instead of `CSWIDTH`) should be used in SunOS 5 systems.

The file `euc.h` has fields for EUC width, screen width, and wide-character width. The functions in Example C-1 are used to set and get EUC widths (these functions assume the environment where the `eucwidth_t` structure is needed and available).

EXAMPLE C-1   EUC Header File

```
#include <eucioctl.h>   /* need others,like stropts.h*/

struct eucioc eucw;   /*for EUC_WSET/WGET to line disc*/
eucwidth_t width;     /* ret struct from _getwidth() */
/*
 * set_euc     Send EUC code widths to line discipline.
 */
set_euc(struct eucioc *e)
{
 struct strioctl sb;

 sb.ic_cmd = EUC_WSET;
 sb.ic_timout = 15;
 sb.ic_len = sizeof(struct eucioc);
 sb.ic_dp = (char *) e;

 if (ioctl(0, I_STR, &sb) < 0)
    fail();
}
/*
 * euclook.   Get current EUC code widths from line discipline.
 */
euclook(struct eucioc *e)
{
 struct strioctl sb;
```

```
sb.ic_cmd = EUC_WGET;
sb.ic_timout = 15;
sb.ic_len = sizeof(struct eucioc);
sb.ic_dp = (char *) e;

if (ioctl(0, I_STR, &sb) < 0)
  fail();

printf("CSWIDTH=%d:%d,%d:%d,%d:%d",
  e->eucw[1], e->scrw[1],
  e->eucw[2], e->scrw[2],
  e->eucw[3], e->scrw[3]);
}
```

Hardware Emulation Module

If a stream supports a terminal interface, a driver or module that understands all `ioctl`s is needed to support terminal semantics (specified by termio(7I) and termiox(7I). If there is no hardware driver that understands all `ioctl` commands downstream from the `ldterm` module, a hardware emulation module must be placed downstream from the line-discipline module. The function of the hardware emulation module is to understand and acknowledge the `ioctl`s that may be sent to the process at the stream head and to mediate the passage of control information downstream. Together, the line-discipline module and the hardware emulation module behave as if there was an actual terminal on that stream.

The hardware emulation module is necessary whenever there is no TTY driver at the end of the stream. For example, the module is necessary in a pseudo-TTY situation where there is process-to-process communication on one system (discussed in "STREAMS-based Pseudo-Terminal Subsystem" on page 312), or in a network situation where a `termio` interface is expected (for example, remote login) but there is no TTY driver on the stream.

Most of the actions taken by the hardware emulation module are the same regardless of the underlying architecture. However, there are some actions that are different, depending on whether the communication is local or remote and whether the underlying transport protocol is used to support the remote connection.

Each hardware emulation module has an `open`, `close`, `read` queue `put` procedure, and write queue `put` procedure.

The hardware emulation module does the following:

■   Processes, if appropriate, and acknowledges receipt of the following `ioctl`s on its write queue by sending an `M_IOCACK` message back upstream: `TCSETA`, `TCSETAW`, `TCSETAF`, `TCSETS`, `TCSETSW`, `TCSETSF`, `TCGETA`, `TCGETS`, and `TCSBRK`.

■   Acknowledges the Extended UNIX Code (EUC) ioctl(2).

- If the environment supports windowing, it acknowledges the windowing `TIOCSWINSZ`, `TIOCGWINSZ`, and `JWINSIZE` `ioctl(2)`s. If the environment does not support windowing, an `M_IOCNAK` message is sent upstream.

- If another `ioctl(2)` is received on its write queue, it sends an `M_IOCNAK` message upstream. It doesn't pass any unrecognized `ioctl`s to the slave driver.

- When the hardware emulation module receives an `M_IOCTL` message of type `TCSBRK` on its write queue, it sends an `M_IOCACK` message upstream and the appropriate message downstream. For example, an `M_BREAK` message could be sent downstream.

- When the hardware emulation module receives an `M_IOCTL` message on its write queue to set the baud rate to `0` (`TCSETAW` with `CBAUD` set to `B0`), it sends an `M_IOCACK` message upstream and an appropriate message downstream; for networking situations this will probably be an M_PROTO message, which is a `TPI T_DISCON_REQ` message requesting the transport provider to disconnect.

- All other messages (`M_DATA`, for instance) not mentioned here are passed to the next module or driver in the stream.

The hardware emulation module processes messages in a way consistent with the driver that exists.

STREAMS-based Pseudo-Terminal Subsystem

The STREAMS-based pseudo-terminal subsystem provides the user with an interface that is identical to the STREAMS-based terminal subsystem described earlier in this chapter. The pseudo-terminal subsystem (pseudo-TTY) supports a pair of STREAMS-based devices called the *master* device and *slave* device. The slave device provides processes with an interface that is identical to the terminal interface. However, where devices that provide the terminal interface have some kind of hardware device behind them, the slave device has another process manipulating it through the master half of the pseudo-terminal. Anything written on the master device is given to the slave as an input, and anything written on the slave device is presented as an input on the master side.

Figure C-2 illustrates the architecture of the STREAMS-based pseudo-terminal subsystem. The master driver (called `ptm`) is accessed through the clone driver and is the controlling part of the system. The slave driver (called `pts`) works with the line discipline module and the hardware emulation module to provide a terminal interface to the user process. An optional packetizing module (called `pckt`) is also provided. It can be pushed on the master side to support packet mode (this is discussed in).

The number of pseudo-TTY devices that can be installed on a system depends on available memory.

Line-Discipline Module

In the pseudo-TTY subsystem, the line discipline module is pushed on the slave side to present the user with the terminal interface.

ldterm(7M) can turn off the processing of the c_iflag, c_oflag, and c_lflag fields to allow processing to take place elsewhere. The ldterm(7M) module can also turn off all canonical processing when it receives an M_CTL message with the MC_NO_CANON command to support remote mode. Although ldterm(7M) passes through messages without processing them, the appropriate flags are set when an ioctl(2), such as TCGETA or TCGETS, is issued to indicate that canonical processing is being performed.

FIGURE C-2 Pseudo-TTY Subsystem Architecture
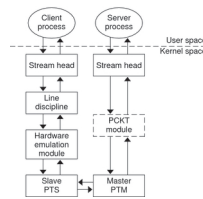
Pseudo-TTY Emulation Module: `ptem`

Because the pseudo-TTY subsystem has no hardware driver downstream from the ldterm(7M) module to process the terminal ioctl(2) calls, another module that understands the ioctl commands is placed downstream from the ldterm(7M). This module, ptem(7M), processes all of the terminal ioctl(2)calls and mediates the passage of control information downstream.

ldterm(7M) and ptem(7M) together behave like a real terminal. Since there is no real terminal or modem in the pseudo-TTY subsystem, some of the ioctl(2) commands are ignored and cause only an acknowledgment of the command. ptem(7M) keeps track of the terminal parameters set by the various set commands such as TCSETA or TCSETAW but does not usually perform any action. For example, if a "set" ioctl is called, none of the bits in the c_cflag field of termio(7I) has any effect on the pseudo-terminal unless the baud rate is set to 0. Setting the baud rate to 0 has the effect of hanging up the pseudo-terminal.

The pseudo-terminal does not recognize parity, so none of the flags in the `c_iflag` that control the processing of parity errors have any effect. The delays specified in the `c_oflag` field are also not supported.

`ptem`(7M) does the following:

- Processes, if appropriate, and acknowledges receipt of the following `ioctls` on its write queue by sending an `M_IOCACK` message back upstream: `TCSETA`, `TCSETAW`, `TCSETAF`, `TCSETS`, `TCSETSW`, `TCSETSF`, `TCGETA`, `TCGETS`, and `TCSBRK`.

- Keeps track of the window size; information needed for the `TIOCSWINSZ`, `TIOCGWINSZ`, and `JWINSIZE ioctl`.

- When it receives an `ioctl`, other than for `TIOCSWINSZ`, `TIOCGWINSZ`, or `JWINSIZE` on its write queue, it sends an `M_IOCNAK` message upstream.

- It passes downstream the following `ioctls` after processing them: `TCSETA`, `TCSETAW`, `TCSETAF`, `TCSETS`, `TCSETSW`, `TCSETSF`, `TCSBRK`, and `TIOCSWINSZ`.

- Frees any `M_IOCNAK` messages it receives on its read queue in case the `pckt` module (`pckt`(7M) (described in the section "Packet Mode" on page 316) is not on the pseudo-terminal subsystem and the `TCSETA`, `TCSETAW`, `TCSETAF`, `TCSETS`, `TCSETSW`, `TCSETSF`, `TCSBRK`, or `TIOCSWINSZioctls` get to the master's stream head which then sends an `M_IOCNAK` message.

- In its open routine, `ptem` sends an `M_SETOPTS` message upstream requesting allocation of a controlling TTY.

- When `ptem` receives an `M_IOCTL` message of type `TCSBRK` on its read queue, it sends an `M_IOCACK` message downstream and an `M_BREAK` message upstream.

- When `ptem` receives an `ioctl`(2) message on its write queue to set the baud rate to 0 (`TCSETAW` with `CBAUD` set to `B0`), it sends an `M_IOCACK` message upstream and a zero-length message downstream.

- When `ptem` receives an `M_IOCTL` of type `TIOCSIGNAL` on its read queue, it sends an `M_IOCACK` downstream and an `M_PCSIG` upstream, where the signal number is the same as in the `M_IOCTL` message.

- When `ptem` receives an `M_IOCTL` of type `TIOCREMOTE` on its read queue, it sends an `M_IOCACK` message downstream and the appropriate `M_CTL` message upstream to enable or disable canonical processing.

- When `ptem` receives an `M_DELAY` message on its read or write queue, it discards the message and does not act on it.

- When `ptem` receives an `M_IOCTL` of type `JWINSIZE` on its write queue, and if the values in its `jwinsize` structure are not zero, it sends an `M_IOCACK` message upstream with the `jwinsize` structure. If the values are zero, it sends an `M_IOCNAK` message upstream.

- When `ptem` receives an `M_IOCTL` message of type `TIOCGWINSZ` on its write queue and the values in the `winsize` structure are not zero, it sends an `M_IOCACK` message upstream with the `winsize` structure. If the values are zero, it sends an `M_IOCNAK` message upstream. It

also saves the information passed to it in the `winsize` structure and sends a STREAMS signal message for signal `SIGWINCH` upstream to the slave process if the size changed.

- When `ptem(7M)` receives an `M_IOCTL` message with type `TIOCGWINSZ` on its read queue and the values in the `winsize` structure are not zero, it sends an `M_IOCACK` message downstream with the `winsize` structure. If the values are zero, it sends an `M_IOCNAK` message downstream. It also saves the information passed to it in the `winsize` structure and sends a STREAMS signal message for signal `SIGWINCH` upstream to the slave process if the size changed.

- All other messages are passed to the next module or driver.

`ptem` Data Structure

Each instantiation of `ptem(7M)` is associated with a local area. These data are held in a structure called `ptem` that has the following format:

```
struct ptem
{
 long cflags;      /* copy of c_flags */
 mblk_t *dack_ptr;   /* pointer to preallocated msg blk
            used to send disconnect */
 queue_t *q_ptr;    /* pointer to ptem's read queue */
 struct winsize wsz;   /*struct to hold windowing info*/
 unsigned short state;  /* state of ptem entry */
};
```

When `ptem(7M)` is pushed onto the slave side stream, a search of the `ptem` structure is made for a free entry (`state` is not set to `INUSE`). The `c_cflags` of the `termio(7I)` structure and the windowing variables are stored in `cflags` and `wsz` respectively. The `dack_ptr` is a pointer to a message block used to send a zero-length message whenever a hang-up occurs on the slave side.

Note - `ptem(7M)` internal implementation might change. This structure should be relevant only to people wanting to change the module.

`open` and `close` Routines

Caution - The following information is implementation-dependent.

In the open routine of `ptem(7M)` a STREAMS message block is allocated for a zero-length message for delivering a hangup message. This allocation of a buffer is done before it is needed

to ensure that a buffer is available. An `M_SETOPTS` message is sent upstream to set the read-side stream head queues, to assign high-water and low-water marks (1024 and 256 respectively), and to establish a controlling terminal.

The same default values as for the line-discipline module are assigned to `cflags`, and `INUSE` to the `state` field.

Note - These default values are currently being examined and may change in the future.

The open routine fails if:

- No free entries are found when the `ptem`(7M) structure is searched
- `sflag` is not set to `MODOPEN`
- A zero-length message cannot be allocated (no buffer is available)
- A `stroptions`(9S) structure cannot be allocated

The `close` routine is called on the last close of the slave-side stream. Pointers to read and write queues are cleared and the buffer for the zero-length message is freed.

Remote Mode

Remote mode available with the pseudo-TTY subsystem, is used for applications that perform the canonical function normally done by `ldterm`(7M) and the TTY driver. The remote mode enables applications on the master side to turn off the canonical processing. An `TIOCREMOTE` `ioctl`(2) with a nonzero parameter (`ioctl(fd, TIOCREMOTE, 1)`) is issued on the master side to enter the remote mode. When this occurs, an `M_CTL` message with the command `MC_NO_CANON` is sent to `ldterm`(7M), indicating that data should be passed when received on the read side and that no canonical processing is to take place. The remote mode may be disabled by `ioctl(fd, TIOCREMOTE, 0)`.

Packet Mode

In the STREAMS-based pseudo-terminal subsystem *packet mode* is used to inform the process on the master side when state changes have occurred in the pseudo-TTY. Packet mode is enabled by pushing the `pckt` module on the master side. Data written on the master side is processed normally. When data is written on the slave side, or when other messages are encountered by the `pckt` module, a header is added to the message so it can be subsequently retrieved by the master side with a `getmsg` operation.

`pckt(7M)` does the following:

- When a message is passed to this module on its write queue, the module does no processing and passes the message to the next module or driver.
- `pckt` creates an `M_PROTO` message when one of the following messages is passed to it: `M_DATA`, `M_IOCTL`, `M_PROTO/M_PCPROTO`, `M_FLUSH`, `M_READ`, `M_START/M_STOP`, and `M_STARTI/M_STOPI`.

  All other messages are passed through. The `M_PROTO` message is passed upstream and retrieved when the user issues `getmsg(2)`.
- If the message is an `M_FLUSH` message, `pckt(7M)` does the following:
  - If the flag is `FLUSHW`, it is changed to `FLUSHR` (because `FLUSHR` was the original flag before the `pts(7D)` driver changed it), changed into an `M_PROTO` message, and passed upstream. To prevent the stream head's read queue from being flushed, the original `M_FLUSH` message must not be passed upstream.
  - If the flag is `FLUSHR`, it is changed to `FLUSHW`, packetized into an `M_PROTO` message, and passed upstream. To flush both of the write queues properly, an `M_FLUSH` message with the `FLUSHW` flag set is also sent upstream.
  - If the flag is `FLUSHRW`, the message with both flags set is packetized and passed upstream. An `M_FLUSH` message with the `FLUSHW` flag set is also sent upstream.

Pseudo-TTY Drivers: `ptm` and `pts`

To use the pseudo-TTY subsystem, a node for the master side driver `/dev/ptmx` and *N* number of slave drivers must be installed (*N* is determined at installation). The names of the slave devices are `/dev/pts/`*M* where *M* has the values 0 through N-1. A user accesses a pseudo-TTY device through the master device (called `ptm`) that in turn is accessed through the clone driver. The master device is set up as a clone device where its major device number is the major for the clone device and its minor device number is the major for the `ptm(7D)` driver.

The master pseudo driver is opened by calling `open(2)` with `/dev/ptmx` as the device to be opened. The clone open finds the next available minor device for that major device. A master device is available only if it, and its corresponding slave device, are not already open. There are no nodes in the file system for master devices.

When the master device is opened, the corresponding slave device is automatically locked out. No user may open that slave device until it is unlocked. A user may invoke a function `grantpt` to change the owner of the slave device to that of the user who is running this process, change the group ID to TTY, and change the mode of the device to 0620. Once the permissions have been changed, the device may be unlocked by the user. Only the owner or the root user can access the slave device. The user must then invoke the `unlockpt` function to unlock the slave device. Before opening the slave device, the user must call the `ptsname` function to obtain the

name of the slave device. The functions `grantpt`, `unlockpt`, and `ptsname` are called with the file descriptor of the master device. The user may then invoke the open system call with the name that was returned by the `ptsname` function to open the slave device.

The following example shows how a user may invoke the pseudo-TTY subsystem:

```
int fdm fds;
char *slavename;
extern char *ptsname();

fdm = open("/dev/ptmx", O_RDWR);  /* open master */
grantpt(fdm);                     /* change permission of slave */
unlockpt(fdm);                    /* unlock slave */
slavename = ptsname(fdm);         /* get name of slave */
fds = open(slavename, O_RDWR);    /* open slave */
ioctl(fds, I_PUSH, "ptem");       /* push ptem */
ioctl(fds, I_PUSH, "ldterm");     /* push ldterm */
```

Unrelated processes may open the pseudo device. The initial user may pass the master file descriptor using a STREAMS-based pipe or a slave name to another process to enable it to open the slave. After the slave device is open, the owner is free to change the permissions.

Note - Certain programs such as `write` and `wall` are set group ID (setgid(2)) to TTY and are also able to access the slave device.

After both the master and slave have been opened, the user has two file descriptors that provide full-duplex communication using two streams. The two streams are automatically connected. The user may then push modules onto either side of the stream. The user also needs to push the `ptem` and `ldterm` modules onto the slave side of the pseudo-terminal subsystem to get terminal semantics.

The master and slave drivers pass all STREAMS messages to their adjacent queues. Only the `M_FLUSH` needs some processing. Because the read queue of one side is connected to the write queue of the other, the `FLUSHR` flag is changed to `FLUSHW` and vice versa.

When the master device is closed, an `M_HANGUP` message is sent to the slave device to render the device unusable. The process on the slave side gets the `errno` `ENXIO` when attempting to write on that stream, but it will be able to read any data remaining on the stream head read queue. When all the data has been read, read(2) returns 0, indicating that the stream can no longer be used.

On the last close of the slave device, a zero-length message is sent to the master device. When the application on the master side issues a `read` or `getmsg` and 0 is returned, the user of the master device decides whether to issue a close that dismantles the pseudo-terminal subsystem. If the master device is not closed, the pseudo-TTY subsystem is available to another user to open the slave device.

Since zero-length messages are used to indicate that the process on the slave side has closed (and should be interpreted that way by the process on the master side), applications on the slave side should not write zero-length messages. If that occurs, the `write` returns 0, and the zero-length message is discarded by the `ptem` module.

The standard STREAMS system calls can access the pseudo-TTY devices. The slave devices support the `O_NDELAY` and `O_NONBLOCK` flags. Because the master side does not act like the terminal, if `O_NONBLOCK` or `O_NDELAY` is set, `read` on the master side returns -1 with `errno` set to `EAGAIN` if no data is available, and `write(2)` returns -1 with `errno` set to `EAGAIN` if there is internal flow control.

The master driver supports the ISPTM and UNLKPT `ioctl(2)` that are used by the functions`grantpt(3C)``unlockpt(3C)`, , and `ptsname(3C)`. The ISPTM `ioctl(2)` determines whether the file descriptor is that of an open master device. On success, it returns the major/minor number (type `dev_t`) of the master device which can be used to determine the name of the corresponding slave device. The UNLKPT `ioctl(2)` unlocks the master and slave devices. It returns 0 on success. On failure, the `errno` is set to `EINVAL` indicating that the master device is not open.

The format of these commands is:

```
int ioctl (int fd, int command, int arg)
```

where *command* is either ISPTM or UNLKPT and *arg* is 0. On failure, -1 is returned.

When data is written to the master side, the entire block of data written is treated as a single line. The slave-side process reading the terminal receives the entire block of data. Data is not edited by the `ldterm` module at input, regardless of the terminal mode. The master-side application is responsible for detecting an interrupt character and sending an interrupt signal SIGINT to the process in the slave side. This can be done as follows:

```
ioctl (fd, TIOCSIGNAL, SIGINT)
```

where SIGINT is defined in the file `signal.h`. When a process on the master side issues this `ioctl(2)`, the argument is the number of the signal that should be sent. The specified signal is then sent to the process group on the slave side.

grantpt

`grantpt(3C)` changes the mode and the ownership of the slave device that is associated with the given master device. Given a file descriptor `fd`, `grantpt(3C)` first checks that the file descriptor is that of the master device. If so, it obtains the name of the associated slave device and sets the user ID to that of the user running the process and the group ID to TTY. The mode of the slave device is set to 0620.

If the process is already running as root, the permission of the slave can be changed directly without invoking this function. `grantpt`(3C) returns 0 on success and -1 on failure. It fails if one or more of the following occurs: `fd` is not an open file descriptor, `fd` is not associated with a master device, the corresponding slave could not be accessed, or a system call failed because no more processes could be created.

unlockpt

`unlockpt`(3C) clears a lock flag associated with a master/slave device pair. `unlockpt`(3C) returns 0 on success and -1 on failure. It fails if one or more of the following occurs: `fd` is not an open file descriptor or `fd` is not associated with a master device.

ptsname

`ptsname`(3C) returns the name of the slave device that is associated with the given master device. It first checks that the file descriptor is that of the master. If it is, it then determines the name of the corresponding slave device `/dev/pts/`*M* and returns a pointer to a string containing the null-terminated path name. The return value points to static data whose content is overwritten by each call. `ptsname`(3C) returns a non-NULL path name upon success and a NULL pointer upon failure. It fails if one or more of the following occurs: `fd` is not an open file descriptor or `fd` is not associated with the master device.

Pseudo-TTY Streams

Drivers and modules can make the stream head act as a terminal stream by sending an `M_SETOPTS` message with the `SO_ISTTY` flag set upstream. This state may be changed by sending an `M_SETOPTS` message with the `SO_ISNTTY` flag set upstream.

Controlling terminals are allocated with the `open`(2) interface. The device must tell the stream head that it is acting as a terminal.

The `TOSTOP` flag is set on reception of an `M_SETOPTS` message with the `SO_TOSTOP` flag set in the `so_flags` field. It is cleared on reception of an `M_SETOPTS` message with the `SO_TONSTOP` flag set.

Stream head processing is isolated from modules and drivers by using several message types, such as `M_ERROR`, `M_HANGUP` and `M_SETOPS`, which only affect the stream in which they are sent.

D

STREAMS FAQ

This appendix provides answers to frequently asked questions (FAQs).

A source of information on STREAMS performance is the paper "The BSD Packet Filter: A New Architecture for User-level Packet Capture" by McCanne & Van Jacobson in the 1993 Winter USENIX proceedings (also available as `ftp://ftp.ee.lbl.gov/papers/bpf-usenix93.ps.Z`). It includes detailed NIT vs. in-kernel BPF performance measurements and some explanation of results obtained.

With decent code in the kernel (not STREAMS) an in-kernel filter is *much* faster.

The following discussion provides answers to frequently asked IP interface questions.

What are the rules for naming a Network Interface Card (NIC) device driver?	Naming of the NIC device driver has the following constraints: ■ The name can contain alphanumeric and underscore (_) characters only ■ The first and last characters of the name cannot be a number ■ The name length cannot exceed 16 characters. The recommended length is 3–8 characters
What is the buffer reclaim policy for Network Interface (NIC) drivers?	Network Interface (NIC) drivers on the transmit side need to reclaim transmit buffers placed to the DMA hardware after the DMA operation has been completed. Because the reclaim operation is expensive, drivers often employ some "lazy-reclaim" scheme to either delay the buffer reclaim, coalesce many completed buffers together in one reclaim call, or both. The most aggressive scheme causes the handling of IO completion and buffer reclaiming to be postponed until a new transmit request is made but no empty slot is left on the DMA hardware.
	Until recently in the Oracle Solaris operating environment, the network transmit buffers have been allocated from the kernel memory heap. The only detrimental effect from lazy reclaim is a waste of kernel memory that could have been freed up sooner.
	The Oracle Solaris operating environment will employ a "zero-copy" scheme whenever possible. Under "zero-copy", the stack will attempt to use client-supplied buffers through `esballoc(9f)` directly for network transmit, rather than allocating kernel heap. The "zero-copy" eliminates

an expensive copy operation when moving data from a client supplied buffer to a kernel buffer.

Lazy reclaim on a client-supply transmit buffer can seriously impact the operation of the client subsystem or application that relies on a buffer to be released from the networking subsystem expediently. For this reason, NIC drivers employing any lazy reclaim scheme must ensure that buffer reclaim is not held up indefinitely. This requirement can be easily satisfied by ensuring that the driver uses a reclaim timer of tens of seconds or less.

Is there documentation that describes the interface between IP and network drivers, namely, the Oracle-specific requirements not outlined in the DLPI Version 2 specification?	IP is a STREAMS module in the Oracle Solaris operating environment. Any module or driver interface with IP should follow the STREAMS mechanism. There are no specific requirements for the interface between IP and network drivers.
When an `ifconfig` device0 plumb is issued, the driver immediately receives a `DL_INFO_REQ`. Exactly what is required in the `DL_INFO_ACK` from a DLPI Style 2 provider?	Look at the `dl_info_ack_t` struct in `/usr/include/sys/dlpi.h`.
Can a driver be a `clone` driver and also a DLPI Style 2 provider? If so, how do I map the minor number selected in the open routine to an instance prior to a `DL_ATTACH_REQ`? The technique of using the minor number to obtain the instance in the `getinfo` routine is	Yes, it is possible for the driver to be a `CLONE` driver and also a DLPI Style 2 provider. The `DL_ATTACH_REQ` request assigns a physical point of attachment (PPA) to a stream. The `DL_ATTACH_REQ` request can be issued any time after a file or stream being opened. The `DL_ATTACH_REQ` request does not have anything to do with assigning, retrieving or mapping minor/instance number. Of course, you can issue a `DL_ATTACH_REQ` request for a file or stream with desired major/minor number. As for the question of mapping minor number to instance, usually the minor number (`getminor(9F)`) is the instance number.

not valid prior to the `DL_ATTACH_REQ`.	
In the examples, a minor node is created each time the driver's attach routine is called. How would a `clone` driver attach to multiple boards; that is, have multiple instances, and still only create one minor node?	For the `clone` driver, this might not be possible. A non-`clone` driver, it can use the bits information in a particular minor number, for example `FF`, to map all other minor nodes.
Do 2.1 Ethernet drivers support LLI 2.0 interfaces?	Do you mean DLPI (Data Link Provider interfaces) ? The Solaris 2.1 ethernet drivers, `le` and `ie`. both support DLPI. See the `le`(7D) man page.
Does Solaris 2.1 DLPI provide both connection-oriented services and connectionless services? Also, is your DLPI Version 2.0, which includes multicast facilities?	Yes and yes. Please see the `dlpi`(7P) man page. TCP and IP are STREAMS modules in the Oracle Solaris operating environment. The command `strconf < /dev/tcp` lists all the modules. STREAMS is not supported in SunOS 4 system TCP/IP.
Is multicasting supported on SunOS 4? If not, how can the customer obtain this feature?	IP multicast is a standard supported feature in the Oracle Solaris operating environment, but it is not supported in the SunOS 4 environment. If customers want to run an unsupported IP multicast on their SunOS 4 machines, it is available via anonymous FTP from `gregorio.stanford.edu` in the file `vmtp-ip/ipmulti-sunos41x.tar.Z`.

Glossary

anchor A STREAMS locking mechanism that prevents the removal of STREAMS modules with the `I_POP` ioctl. Anchors are placed on STREAMS modules by adding the `[anchor]` flag to `autopush` configuration files or by directly calling the `I_ANCHOR` ioctl.

autopush A STREAMS mechanism that enables a prespecified list of modules to be pushed automatically onto the stream when a STREAMS device is opened. This mechanism is used only for administrative purposes.

back-enable To enable (by STREAMS) a preceding blocked queue's service procedure when STREAMS determines that a succeeding queue has reached its low-water mark.

blocked A queue's service procedure that cannot be enabled due to flow control.

clone device A STREAMS device that returns an unused major/minor device when initially opened, rather than requiring the minor device to be specified by name in the open call.

close routine A procedure that is called when a module is popped from a stream or when a driver is closed.

controlling stream A stream above the multiplexing driver used to establish the lower connections. Multiplexed stream configurations are maintained through the controlling stream to a multiplexing driver.

DDI Device Driver Interface. An interface that facilitates driver portability across different UNIX system versions.

device driver A stream component whose principle functions are handling an associated physical device and transforming data and information between the external interface and stream.

DKI Driver–Kernel Interface. An interface between the UNIX system kernel and different types of drivers. It consists of a set of driver-defined functions that are called by the kernel. These functions are entry points into a driver.

downstream A direction of data flow going from the stream head toward a driver. Also called write side and output side.

driver A module that forms the stream end. It can be a device driver or a pseudo-device driver. It is a required component in STREAMS (except in STREAMS-based pipe mechanism), and physically identical to a module. It typically handles data transfer between the kernel and a device and does little or no processing of data.

enable　　　A term used to describe scheduling of a queue's service procedure.

FIFO　　　First in, first out. A term used in STREAMS for named pipes. This term is also used in queue scheduling.

flow control　　　A STREAMS mechanism that regulates the rate of message transfer within a stream and from user space into a stream.

hardware
emulation
module　　　A module required when the terminal line discipline is on a stream but there is no terminal driver at the end of the stream. This module recognizes all `ioctls` necessary to support terminal semantics specified by `termio` and `termios`.

input side　　　A direction of data flow going from a driver toward the stream head. Also called read side and upstream.

line discipline　　　A STREAMS module that performs `termio` canonical and non-canonical processing. It shares some `termio` processing with a driver in a STREAMS terminal subsystem.

lower stream　　　A stream connected below a multiplexer pseudo-device driver, by means of an `I_LINK` or `I_PLINK` ioctl. The far end of a lower stream terminates at a device driver or another multiplexer driver.

master driver　　　A STREAMS-based device supported by the pseudo-terminal subsystem. It is the controlling part of the pseudo-terminal subsystem (also called `ptm`).

message　　　One or more linked message blocks. A message is referenced by its first message block and its type is defined by the message type of that block.

message block　　　A triplet consisting of a data buffer and associated control structures, an `msgb` structure and a `datab` structure. It carries data or information, as identified by its message type, in a stream.

message
queue　　　A linked list of zero or more messages connected together.

message type　　　A defined set of values identifying the contents of a message.

module　　　A defined set of kernel-level routines and data structures used to process data, status, and control information on a stream. It is an optional element, but there can be many modules in one stream. It consists of a pair of queues (read queue and write queue), and it communicates to other components in a stream by passing messages.

multiplexer　　　A STREAMS mechanism that enables messages to be routed among multiple streams in the kernel. A multiplexing configuration includes at least one multiplexing pseudo-device driver connected to one or more upper streams and one or more lower streams.

named stream　　　A stream, typically a pipe, with a name associated with it by way of a call to `fattach` (that is, a mount operation). This is different from a named pipe (FIFO) in two ways: a named pipe

(FIFO) is unidirectional while a named stream is bidirectional; a named stream need not refer to a pipe but can be another type of a stream.

open routine   A procedure in each STREAMS driver and module called by STREAMS on each `open` system call made on the stream. A module's `open` procedure is also called when the module is pushed.

packet mode   A feature supported by the STREAMS-based pseudo-terminal subsystem. It is used to inform a process on the master side when state changes occur on the slave side of a pseudo-TTY. It is enabled by pushing a module called `pckt` on the master side.

persistent link   A connection below a multiplexer that can exist without having an open controlling stream associated with it.

pipe   See STREAMS-based pipe.

pop   A term used when a module that is immediately below the stream head is removed.

pseudo-device driver   A software driver, not directly associated with a physical device, that performs functions internal to a stream such as a multiplexer or log driver.

pseudo-terminal subsystem   A user interface identical to a terminal subsystem except that there is a process in place of a hardware device. It consists of at least a master device, slave device, line-discipline module, and hardware emulation module.

push   A term used when a module is inserted in a stream immediately below the stream head.

pushable module   A module put between the stream head and driver. It performs intermediate transformations on messages flowing between the stream head and driver. A driver is a non-pushable module.

put procedure   A routine in a module or driver associated with a queue that receives messages from the preceding queue. It is the single entry point into a queue from a preceding queue. It may perform processing on the message and will then generally either queue the message for subsequent processing by this queue's service procedure, or will pass the message to the `put` procedure of the following queue.

queue   A data structure that contains status information, a pointer to routines processing messages, and pointers for administering a stream. It typically contains pointers to a `put` and `service` procedure, a message queue, and private data.

read queue   A message queue in a module or driver containing messages moving upstream. Associated with the `read` system call and input from a driver.

read side   A direction of data flow going from a driver towards the stream head. Also called upstream and input side.

remote mode   A feature available with the pseudo-terminal subsystem. It is used for applications that perform the canonical and echoing functions normally done by the line discipline module and TTY driver. It enables applications on the master side to turn off the canonical processing.

SAD   A STREAMS Administrative Driver that provides an interface to the `autopush` mechanism.

schedule   To place a queue on the internal list of queues that will subsequently have their service procedure called by the STREAMS scheduler. STREAMS scheduling is independent of the Oracle Solaris operating environment process scheduling.

service interface   A set of primitives that define a service at the boundary between a service user and a service provider and the rules (typically represented by a state machine) for allowable sequences of the primitives across the boundary. At a stream/user boundary, the primitives are typically contained in the control part of a message; within a stream, in `M_PROTO` or `M_PCPROTO` message blocks.

service procedure   A routine module or driver associated with a queue that receives messages queued for it by the `put` procedure of that queue. The procedure is called by the STREAMS scheduler. It may perform processing on the message and generally passes the message to the `put` procedure of the following queue.

service provider   An entity in a service interface that responds to request primitives from the service user with response and event primitives.

service user   An entity in a service interface that generates request primitives for the service provider and consumes response and event primitives.

slave driver   A STREAMS-based device supported by the pseudo-terminal subsystem. It is also called `pts` and works with a line discipline module and hardware emulation module to provide an interface to a user process.

standard pipe   A mechanism for the unidirectional flow of data between two processes where data written by one process becomes data read by the other process.

stream   A kernel aggregate created by connecting STREAMS components, resulting from an application of the STREAMS mechanism. The primary components are the stream head, the driver, and zero or more pushable modules between the stream head and driver.

stream end   A stream component furthest from the user process that contains a driver.

stream head   A stream component closest to the user process. It provides the interface between the stream and the user process.

STREAMS   A kernel mechanism that provides the framework for network services and data communication. It defines interface standards for character input/output within the kernel, and between the kernel and user level. The STREAMS mechanism includes integral functions, utility routines, kernel facilities, and a set of structures.

STREAMS-based pipe   A mechanism used for bidirectional data transfer implemented using STREAMS, and sharing the properties of STREAMS-based devices.

TTY driver   A STREAMS-based device used in a terminal subsystem.

upper stream   A stream that terminates above a multiplexer. The beginning of an upper stream originates at the stream head or another multiplexer driver.

upstream   A direction of data flow going from a driver towards the stream head. Also called read-side and input side.

water mark   A limit value used in flow control. Each queue has a high-water mark and a low-water mark. The high-water mark value indicates the upper limit related to the number of bytes contained on the queue. When the queued character reaches its high-water mark, STREAMS causes another queue that attempts to send a message to this queue to become blocked. When the characters in this queue are reduced to the low-water mark value, the other queue is unblocked by STREAMS.

write queue   A message queue in a module or driver containing messages moving downstream. Associated with the `write` system call and output from a user process.

write side   A direction of data flow going from the stream head toward a driver. Also called downstream and output side.

Index