STREAMS Programming Guide

Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043-1100
U.S.A.

Part No: 802-5893
August 1997

Please Recycle

Contents

Preface

The *STREAMS Programming Guide* describes how to use STREAMS in designing and implementating applications and STREAMS modules and drivers, for architectures that conform to the Solaris™ 2.6 DDI/DDK.

Who Should Read This Book

The 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*.

How This Book Is Organized

This guide is divided into three parts. Part 1 describes how to use STREAMS facilities in applications. Part 2 describes how to design STREAMS modules and STREAMS drivers. Part 3 contains advanced topics. Every developer should read Chapter 1.

- Chapter 1 is a general overview of STREAMS concepts and mechanisms.
- Chapter 2 describes the basic operations to assemble, use, and dismantle Streams.
- Chapter 3 details the operations of messages, the flow of Streams, and how to manipulate Steams from applications.

- Chapter 4 describes putting messages into and receiving them from a Stream..
- Chapter 5 identifies and describes tools to monitor names and modules, and gather statistics.
- Chapter 6 describes pipes and named pipes (FIFOs).

- Chapter 7 describes STREAMS modules, drivers, and how they relate.
- Chapter 8 describes message types, structure, and linkage in detail. Flow control is also covered.
- Chapter 9 describes specific STREAMS drivers, using code samples.
- Chapter 10 describes how specific examples of modules work based on code samples.
- Chapter 11 describes configuring modules and drivers into the OS.
- Chapter 12 describes the multithreaded environment and how to make modules and drivers MT- safe.
- Chapter 13 describes how to implement multiplexing in a driver.

- Chapter 14 explains setting up a terminal subsystem, and keeping track of processes and handling interrupts.
- Chapter 15 addresses commonly encountered problems and their resolution.
- Appendix C examines changes you can make to get the maximum performance out of STREAMS drivers and modules.

Appendices

- Appendix A describes STREAMS messages and their use.
- Appendix B describes STREAMS utility routines and their use.
- Appendix C contains answers to a variaty of commonly asked questions about STREAMS.
- *Glossary* defines terms unique to STREAMS.

Code Examples

All code examples used in this book conform to ANSI C specifications.

Conventions Used

The word “STREAMS” refers to the mechanism and “Stream” refers to an explicit path between a user application and a driver.

Examples highlight common capabilities of STREAMS, and reference fictional drivers and modules. Where possible, examples are executable code.

The following table describes the typographic conventions used in this book.

TABLE P–1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% You have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`** `Password:`
AaBbCc123	Interface or command-line variable: provide a real path or value	To delete a file, type `rm` *filename.*
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User’s Guide.* These are called *class* options. You *must* be root to do this.

Warning - The warning sign shows possible damage to data, system, application, or person.

Caution - The caution sign shows possible harm or damage to a system, an application, a process, or a piece of hardware.

Note - Notes are used to emphasize points of interest, to present parenthetical information, and to cite references to other documents and commands.

Related Books

You can obtain more information on STREAMS system calls and utilities from the online manual pages. For more information on driver-related issues, including autoconfiguration, see *Writing Device Drivers.*

You can also find STREAMS described to some extent in the System V Interface Definition, and in the following publications:

Goodheart, Berny and Cox, James. *The Magic Garden Explained.*Australia, & Englewood Cliffs, New Jersey: Prentice Hall, 1994.

Rago, Stephen A. *UNIX System V Network Programming.* Reading, Massachusetts: Addison-Wesley, 1993.

PART I Application Programming Interface

CHAPTER 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. Since the application developer is concerned with a different subset of STREAMS interfaces than the kernel-level developer, application and kernel levels are described separately.

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 lets you create modules to provide standard data communications services and 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 STREAMS driver in kernel space. A Stream has three parts: a Stream head, zero or more modules, and a driver.

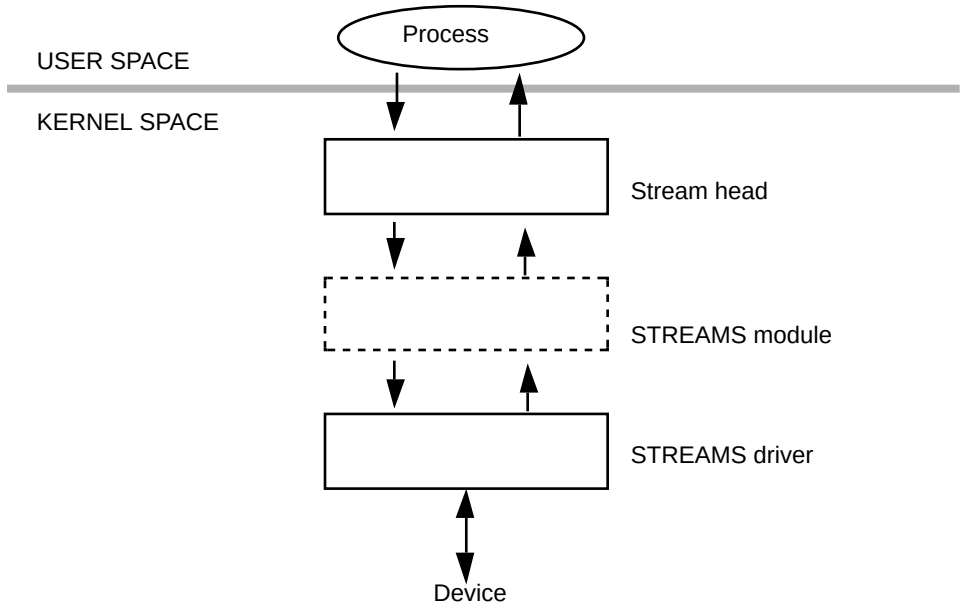


Figure 1–1 Simple Stream

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

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.

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 user level by an application. Full details on modules and their operation are covered in Chapter 10.

Driver

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 Solaris 2.x Device Driver Interface/Driver Kernel Interface (Solaris 2.x DDI/DKI). The relationship between the driver and the rest of the UNIX kernel is explained in *Writing Device Drivers*. Details of device drivers are explained in Chapter 9.

Messages

The means by which all I/O is done under STREAMS. Data on a Stream is passed in the form of messages. 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 on “Messages” on page 13.

Queues

A container for messages. 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 on “Queues” on page 15.

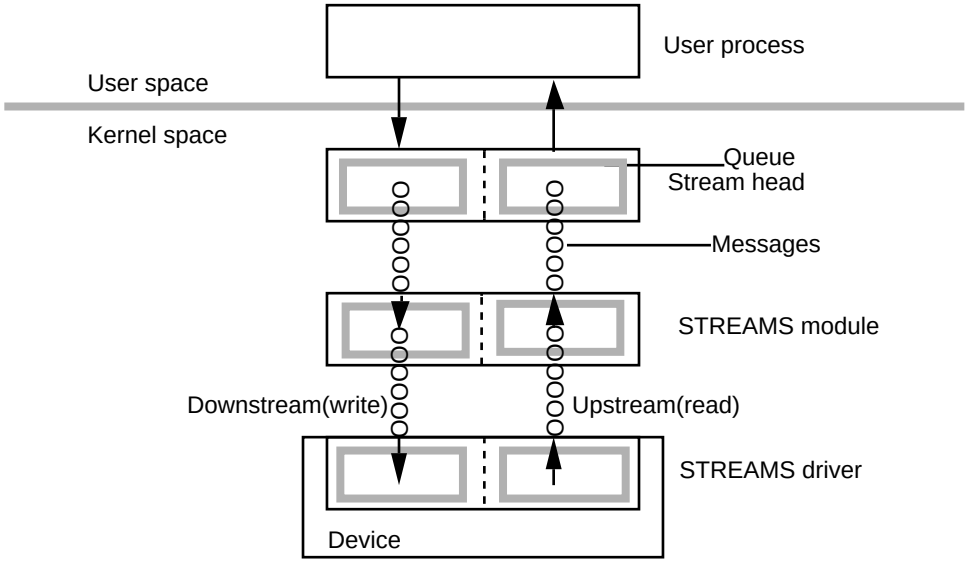


Figure 1–2 Messages Passing Using Queues

streamio

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 details Inter-Stream communications.

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 start on “Multiplexing” on page 16.

Polling

Polling within STREAMS allows 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 allows applications to detect events that occur at the head of one or more Streams. Chapter 3 describes polling.

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 with data. Flow control is local to each module or driver and voluntary. Chapter 8 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, since 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 describes properties and behavior of clone devices.

Once a device is opened, a user process can send data to the device by calling **wait**(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 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

Code Example 1–1shows 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.

CODE 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 1–3shows the state of the Stream following the call to **open**(2).

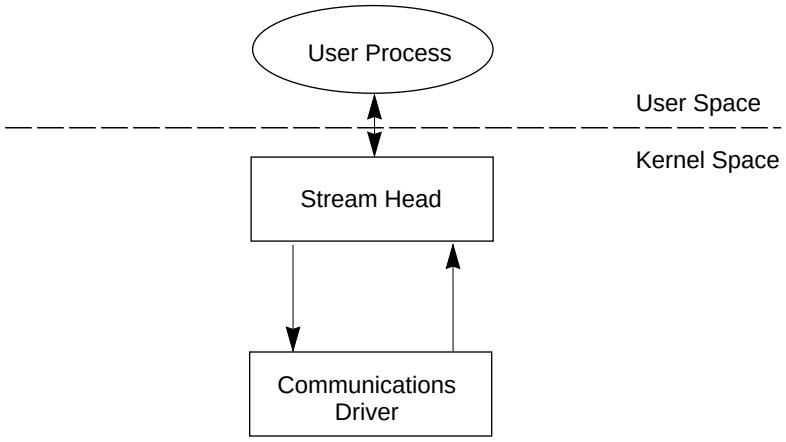


Figure 1–3 Stream to Communications Driver

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—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.

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.

Modules

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 1–4shows 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 down stream queues only if the module functions are specifically programed 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.

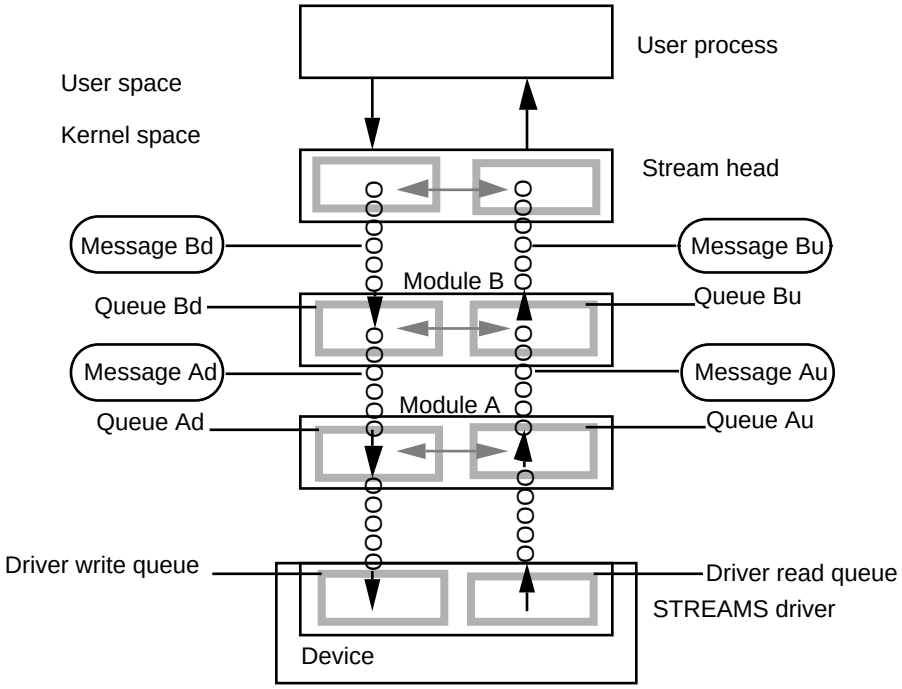


Figure 1–4 Stream in More Detail

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 procedures — Individual `put` and `service` routines on the read and write queues 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 reopened. The `close` is

invoked when the module is popped or the Stream 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)

Drivers

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.

Messages

All kernel-level input and output under STREAMS is based on messages. STREAMS messages are built in triplets: a message header (**msgb**(9S)) that contains information pertinent to the message instance; a data block (**datab**(9S)) that contains information describing the data; and 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.

STREAMS messages use two data structures (`msgb`– the message header, and datab– the data block) to describe the message data. These data structures identify the type of message and point to the data of the message, plus other information. 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 lets developers manipulate messages and message queues.

Message Types

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 that enable support of urgent, expedited data, are placed in the queue after high-priority messages but before ordinary messages. Priority band messages are placed below all messages that have a priority greater than or equal to their own, but above any with a lesser priority. Figure 1-5 shows the message queueing priorities

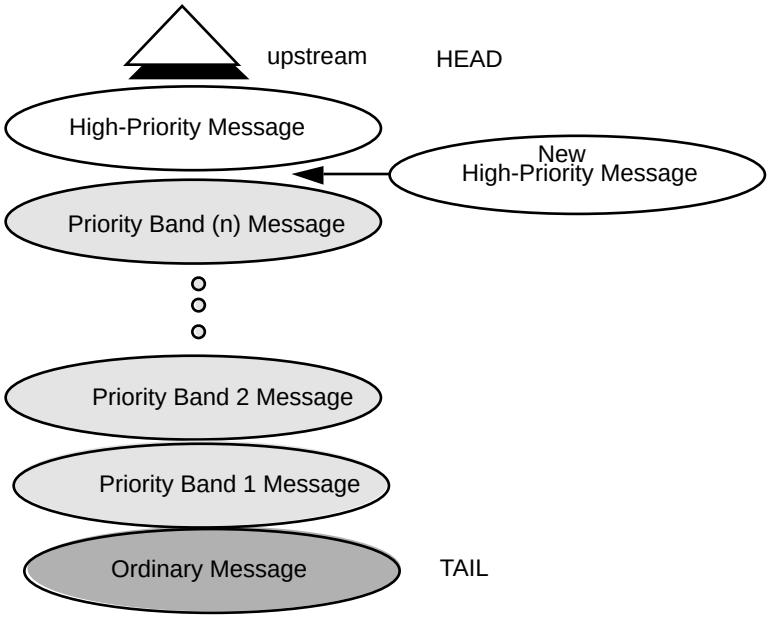


Figure 1–5 Message Priorities

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.

Queues

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 7discusses 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.

Multiplexing

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 (which 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 in the three previous figures 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 multiplexor 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. It is up to the implementation how data is passed between the upper and lower multiplexer. Chapter 13” covers implementing multiplexers.

Multithreading

The Solaris 2.x 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 allows the developer control over the level of concurrency allowed in a module. The DDI/DKI provides several advisory locks for protecting data. See Chapter 12, for more information.

STREAMS in Operation

Service Interfaces

STREAMS makes it possible to 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 sends 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 only be connected to the input and output side if they have the compatible X.25 service interface.

Manipulating Modules

With STREAMS you can manipulate modules from 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 the same 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).

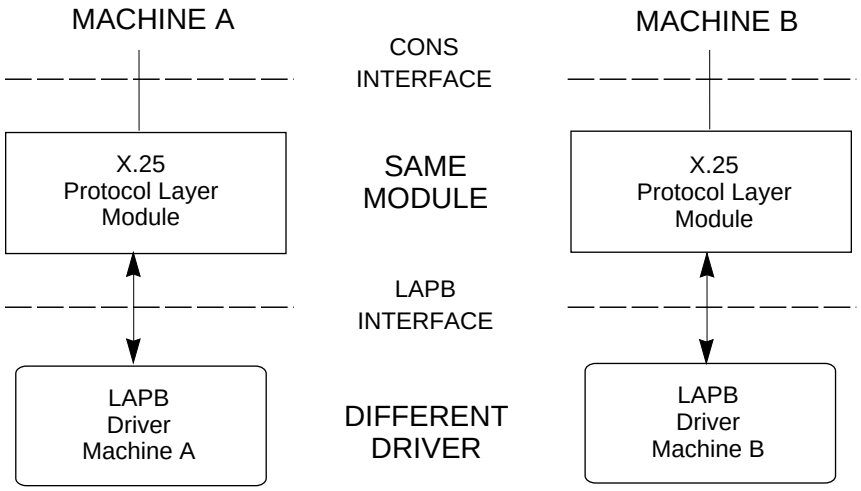


Figure 1–6 Protocol Module Portability

Protocol Substitution

You can use 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.

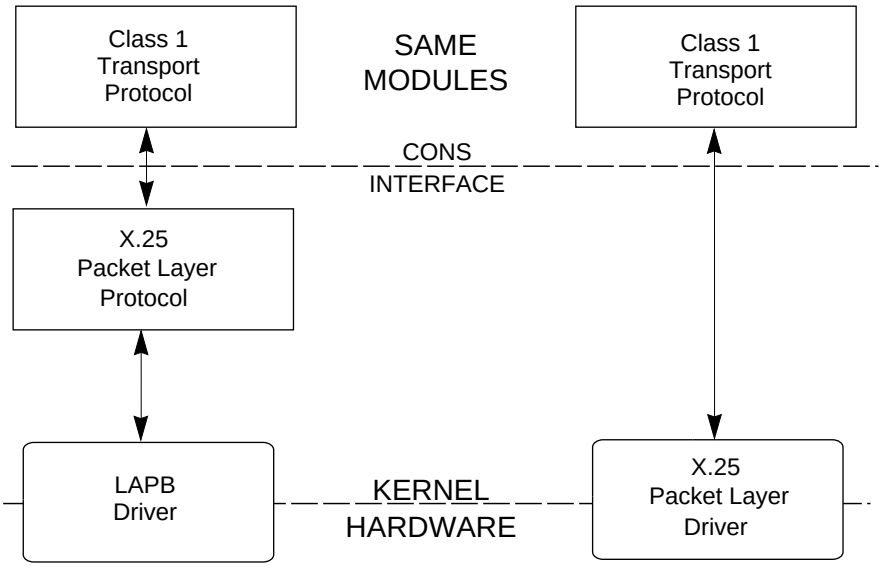


Figure 1–7 Protocol Migration

Module Reusability

Figure 1–8shows 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 since the module is nearest the Stream head.

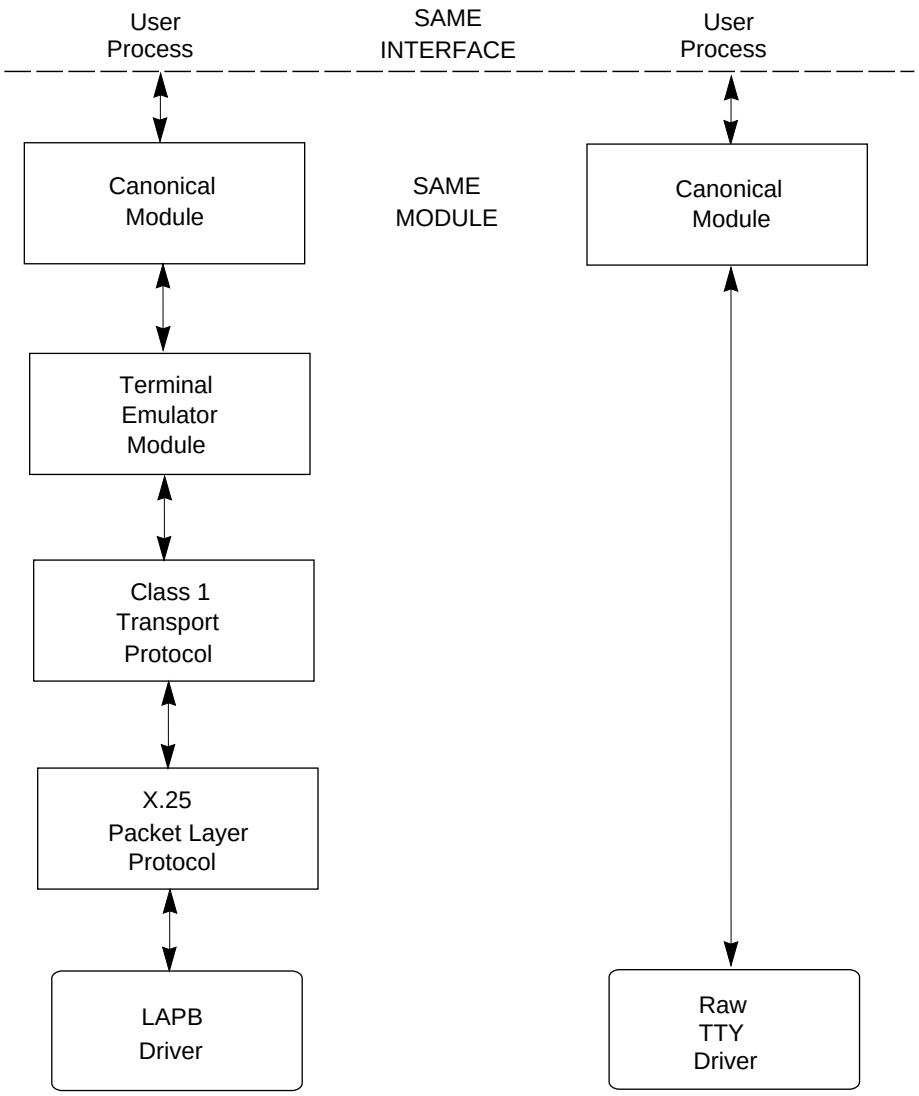


Figure 1–8 Module Reusability

CHAPTER 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 example.

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 lets a user process 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

manpage	Description
open(2)	Opens a Stream
close(2)	Closes a Stream
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 (not restricted to STREAMS, although historically 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 on line 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 **streamio**(7I)) 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 **open**(2) 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 simply returns a unique minor number.

When a Stream is already open, further opens of the same device result in the open routines of all modules and the driver on the Stream being called. 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 **autopush**(1M) and **sad**(7D).

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.

Queueing

STREAMS queues are allocated in pairs. One queue is always the upstream or read-side; the other is the downstream, 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). Then, the Stream head 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.

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`, see **`open`**(2) and **`fcntl`**(2)) is clear, **`close`**(2) waits up to 15 seconds for each module to drain and up to 15 seconds for the driver to drain. The default close delay is 15 seconds, but this can be changed on a per-stream basis with the `I_SETCLTIME` **`ioctl`**(2).

Closing Delay Details

The 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 9. The module in this example converts (change case, delete, duplicate) selected alphabetic characters.

Note - The complete listing of the module is on the CD....(indicate correct name here.)

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. Code fragments with explanations follow in Code Example 2–1.

CODE EXAMPLE 2–1 Header Definition

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

The first step is to establish a Stream to the communications driver and insert the character conversion module. 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 Code Example 2–2.

CODE 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.

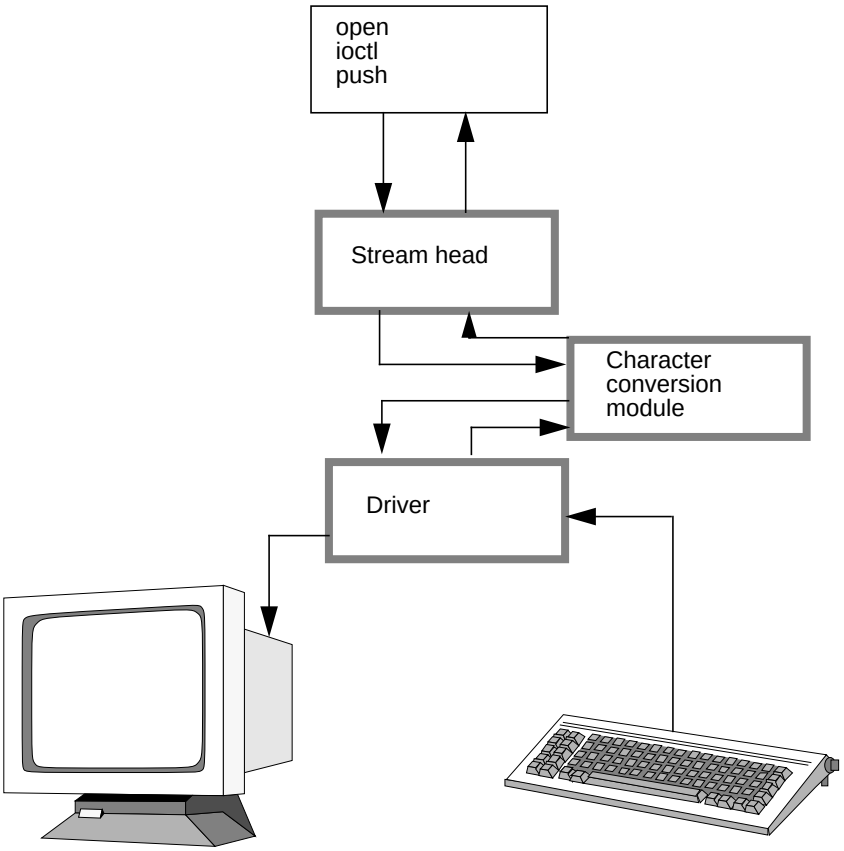


Figure 2–1 Pushing the Character Conversion Module

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 as shown in the next example.

Code Example 2–3 uses the conventional I_STR **ioctl**(2) an indirect way of passing commands and data pointers. Code 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 which also accept theI_STR form.

CODE 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 Code 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 Code Example 2–4.

CODE EXAMPLE 2–4 Structure of strioctl

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

where:

ic_cmd — identifies the command intended for a module or driver.

`ic_timout`	specifies the number of seconds an I_STR request should wait for an acknowledgment before timing out.
`ic_len`	is the number of bytes of data to accompany the request.
`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, and 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 acknowledgment message. If an acknowledgment is not received in `ic_timout` seconds, **`ioctl`**(2) fails.

Note - Only one **`ioctl`**(2) can be active on a Stream at one time, whether or not it is issued with `I_STR`. Further requests will block until the active **`ioctl`**(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` will be set on return to indicate the amount of data returned.

The remainder of this example is identical to Code Example 1–1 in Chapter 1.

CODE 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) 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.

CHAPTER 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

Since 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 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 `service` procedure. For a more detailed explanation of `put` and `service`, see Chapter 8.

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 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, which are discussed in. Packet sizes are discussed here. Watermarks are discussed in “Flush Handling” on page 62 in Chapter 4.

Read Options

The value of read options (`so_readopt`) specifies 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.

TABLE 3–1 Read Option Modes

RMODEMASK		
RNORM	RMSGN	RMSGN
RPROTMASK		
RPROTNORM	RPROTDIS	RPROTDATA

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

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:

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.

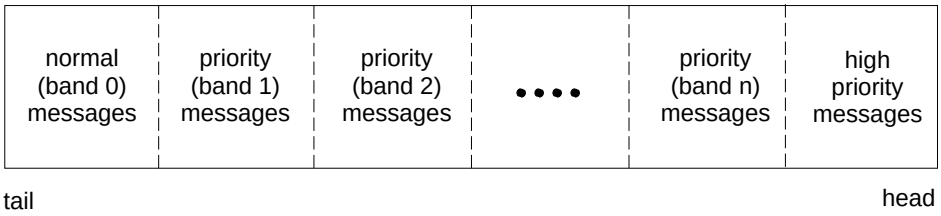


Figure 3–1 Message Ordering in a Queue

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.

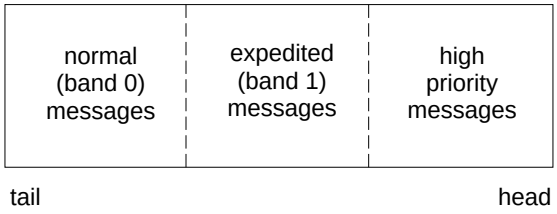


Figure 3–2 Message Ordering With One Priority Band

Controlling 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 62 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 Transmission Control Protocol’s (TCP) ability 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 if the message on the front of its Stream-head read queue is marked or not 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. Get the rest of the data with successive reads. Chapter 4discusses 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 merely wants to check if any message is marked. LASTMARK indicates that the user wants to see if 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. Code Example 3–1 shows the `inter_open` routine.

CODE EXAMPLE 3–1 `inter_open` Routine

```
inter_open (char *path, oflags, addr)
{
  intt 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 acknowledgment from the service provider, as Code Example 3–2 shows:

CODE 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 acknowledgment of the bind request. The acknowledgment message consists of a control part that contains either an `OK_ACK` or an `error_ack` structure, and no data part.

The acknowledgment 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; high-priority messages are placed at the front of the Stream-head queue, followed by priority band messages and ordinary messages. The STREAMS mechanism allows only one high-priority message per Stream at the Stream-head at one time. Any additional high-priority messages is 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 acknowledgment 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 acknowledgment 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. It is set because you want to catch the acknowledgment 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 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.

```
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 acknowledgment 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 Code Example 3–3, `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.

CODE EXAMPLE 3–3 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 nondiscard 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 for brevity. 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 allow 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 process(es) 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`, detailed in Table 3-2.

TABLE 3–2 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 Solaris System.
`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).

TABLE 3–2 Events That Can Be Polled *(continued)*

Event	Description
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, it is not expected that the POLLPRI event is 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).

Code Example 3–4 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.

CODE EXAMPLE 3–4 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 3-3 shows poll error events.

TABLE 3–3 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 can't be writable if a hangup has occurred.
POLLNVAL	The specified file descriptor is not associated with an open Stream.

These events can not 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 were opened with the O_NDELAY (or O_NONBLOCK; see note) flag set. **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, whose 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) lets a user monitor multiple Streams synchronously. **poll**(2) normally blocks until an event occurs on any of the polled file descriptors. In some applications, however, it is desirable 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 3-4 lists events for I_SETSIG. These are similar to those described for **poll**(2).

TABLE 3–4 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 Solaris.
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.
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.

TABLE 3–4 I_SETSIG **ioctl**(2) Events *(continued)*

Event	Description
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.

Signals

STREAMS allows modules and drivers to cause a signal to be sent to user processes through a special signal message. If the signal specified by the module or driver is not SIGPOLL (see **signal**(5)), 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

To let a process 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 3-5 describes the data available in the siginfo_t structure passed to the signal handler.

TABLE 3–5 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
S_MSG	SIGPOLL	POLL_MSG	Band signaled	Unused

TABLE 3–5 Data in `siginfo_t` Structure *(continued)*

Event	si_signo	si_code	si_band	si_errno
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 not only receive signals but also 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`**(5), **`sigsend`**(2), **`termios`**(3), **`waitid`**(2), **`waitpid`**(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. One job, the foreground job, is given complete access to the controlling terminal. The other jobs, 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 can not 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`**(5))

`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 one 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 terminalsare 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 "Accessing the Controlling Terminal" on page 51.)

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

A Stream is allocated as a controlling terminal for a session if:

- The Stream is acting as a terminal.
- The Stream is not already allocated as a controlling terminal.
- The Stream 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, since 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

CHAPTER 4

STREAMS Driver and Module Interfaces

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

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

manpage	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 (not restricted to streams, although historically was)
pipe(2)	Creates a bidirectional channel that provides a communication path between multiple processes

Module and Driver **ioctl**(2)s

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)s 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)s 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. (It is likely that by the time the module or driver receives the **ioctl**(2), the process generating it may no longer be running.)

A third difference is that for the character I/O mechanism, all **ioctl**(2)s 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)s. That is, the **ioctl**(2)s 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)s generated by applications that are implemented for a non-STREAMS environment.

General **ioctl**(2) 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 acknowledgment) message or an M_IOCNAK (nak, negative acknowledgment) 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)s, the default value (infinite) is used.

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

A module that receives an unrecognized M_IOCTL </function> message must pass it on unchanged. A driver that receives an unrecognized M_IOCTL must produce a negative acknowledgment.

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. There is nothing preventing a given **ioctl**(2) from being issued either directly (transparent) or by means of I_STR. Furthermore, **ioctl**(2)s issued through I_STR potentially can require further processing of the form typically associated with transparent **ioctl**(2)s.

I_STR **ioctl**(2) 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 stioctl structure. The module can also return data to this block.

Transparent ioctl(2) Processing

The transparent STREAMS **ioctl**(2) mechanism allows application programs to perform module and driver control functions with **ioctl**(2)s other than I_STR. It is

intended to transparently support 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, it relieves applications of the burden of packaging 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)s are known as transparent **ioctl**(2)s to differentiate them from the I_STR form. Existing applications which use non-STREAMS character devices require transparent processing for **ioctl**(2)s 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**(2)

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 superuser 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 Code Example 4–1) performs two functions. When the third argument of the **ioctl**(2) call is NULL [line 23], 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_list` 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 (see note) 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.

Note -

CODE EXAMPLE 4–1 `I_LIST` **ioctl**(2)

```
#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`**(2) Commands

`streamio`(7I) details the following **`ioctl`**(2) commands.

TABLE 4–2 Other **`ioctl`**(2)s

`ioctl(2)`	**Description**
`I_LOOK`	Retrieves the name of the module just below the Stream head
`I_LOOK`	Flushes all input or output queues
`I_LOOKBAND`	Flushes a band of messages

TABLE 4–2 Other **ioctl**(2)s *(continued)*

ioctl(2)	**Description**
I_FIND	Compares the names of all modules present in the Stream to a specific name
I_PEEK	Lets the user look at information in the first message on the Stream head read queue without taking the message off the queue
I_STRDOPT	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 Stream pipe
I_RECVFD	Retrieves the file descriptor of the message sent by an I_SENDFD **ioctl**(2) over a Stream 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 he Stream-head read queue
I_GETNBAND	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	Lets the user 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

TABLE 4–2 Other **ioctl**(2)s *(continued)*

ioctl(2)	**Description**
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 to prioritize the messages.

TABLE 4–3 Send and Recieve Mesages

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 lets the caller 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 lets the caller 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_LOOK **ioctl**(2). Data may be flushed from the read side, write side, or both sides of a Stream.

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

Table 4–4 describes the arguments that may be passed.

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

Flushing Priority Bands

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

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

The **ioctl**(2) is passed pointer to a bandanna. The beeper field indicates the band priority to be flushed. This may be from 0 to 255. The bi_flag field is used to indicate the type of flushing to be done. The legal values for bi_flag are defined in Table 4-3. bandanna has the following format:

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

See “Flushing Priority Band” on page 141 which describes M_FLUSHBAND processing for details on how modules and drivers should handle flush band requests.

CHAPTER 5

STREAMS Administration

This chapter describes the tools available to administer STREAMS. It attempts to show how to keep track of names, where to find modules and how to monitor statistics. Kernel debugging is covered in Chapter 15.

Tools Available

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

TABLE 5–1 Tools Available for STREAMS Administration

manpage	Description
autopush(1M)	Configures list of modules to be automatically pushed
crash(1M)	Examines system memory images
fdetach(1M)	Detaches a name from a file descriptor

TABLE 5–1 Tools Available for STREAMS Administration *(continued)*

manpage	Description
strace(1M)	Prinst STREAMS trace messages
strace(1M)	Changes or queries a Stream configuration
strerr(1M)	Logs STREAMS errors
modload(1M)	Loads a kernel module
modunload(1M)	Unloads a kernel module
modinfo(1M)	Displays information about loaded kernel modules

Autopush Facility

The autopush facility (see **autopush**(1M)) lets administrators 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.

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.

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 superuser 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 */
  long apc_major;     /* major device number */
  long apc_minor;     /* minor device number */
  long 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_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`. 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`:

```
# /dev/console and /dev/contty autopush setup
#
# 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(1M)

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(9F)

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**(3S)-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(9F)

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(9F)

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(1M)

strerr(1M) is the streams error logger daemon.

CHAPTER 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 is a mechanism that provides a communication path between multiple processes. Prior to SunOS 5.0, SunOS had *standard* pipes and named pipes (also called FIFOs, or first-in first-out). With standard pipes, one end was opened for reading and the other end for writing, thus 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 have become STREAMS-based in SunOS 5.x. The interface is unchanged but the underlying implementation has 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 for a STREAMS-based pipe.

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**(3C), 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 the 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.

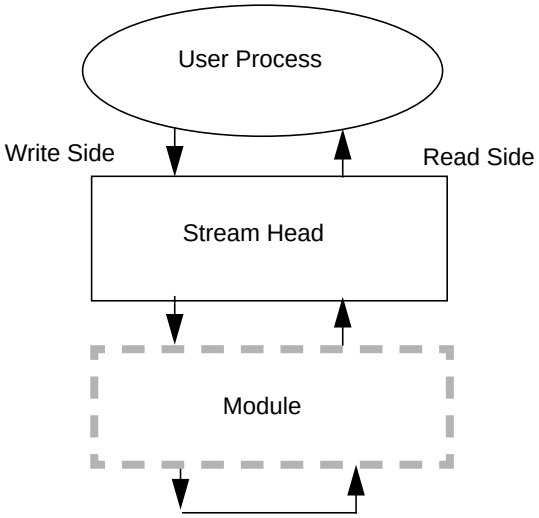


Figure 6–1 Pushing Modules on a STREAMS-based FIFO

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 are 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 74 for details.

A STREAMS-based pipe, also referred to as an anonymous pipe, is created using **pipe**(2). **pipe**(2) returns two file descriptors—fd[0] and fd[1], each with its own Stream head. The ends of the pipe are so constructed 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.

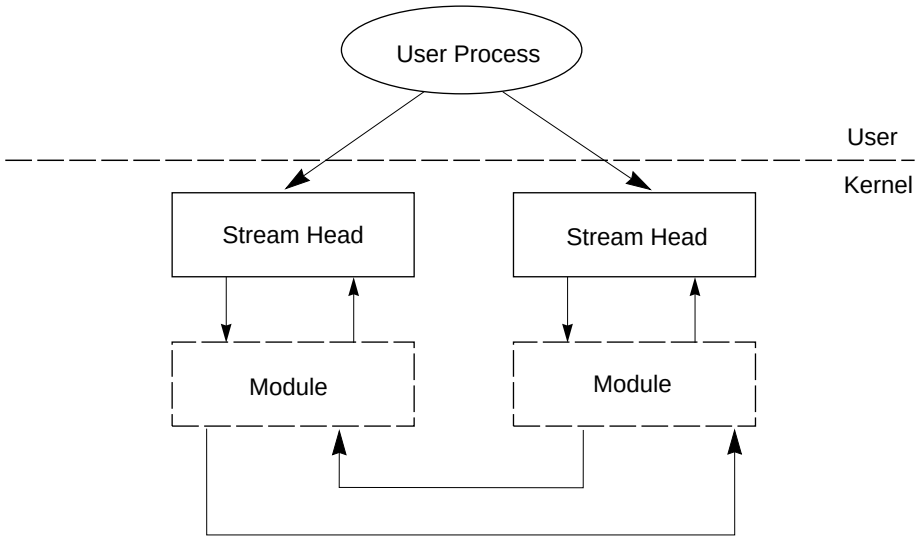


Figure 6–2 Pushing Modules on a STREAMS-based Pipe

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), getpmesg(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), data 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 un`read` 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 sends 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 written is not necessarily the order of data 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 and/or the FLUSHW bits of an M_FLUSH message must be switched. The point of switching the bits is 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 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 via **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

It can be useful to name a Stream or STREAMS based pipe by associating 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 interfaces identified below support naming a Stream or STREAMS based pipe.

fattach(3C)
fdetach(3C)
isastream(3C)

Passing File Descriptors

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 a 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 module connld that lets a client process get a unique, non-multiplexed connection to a server. 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 illustrates 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`.

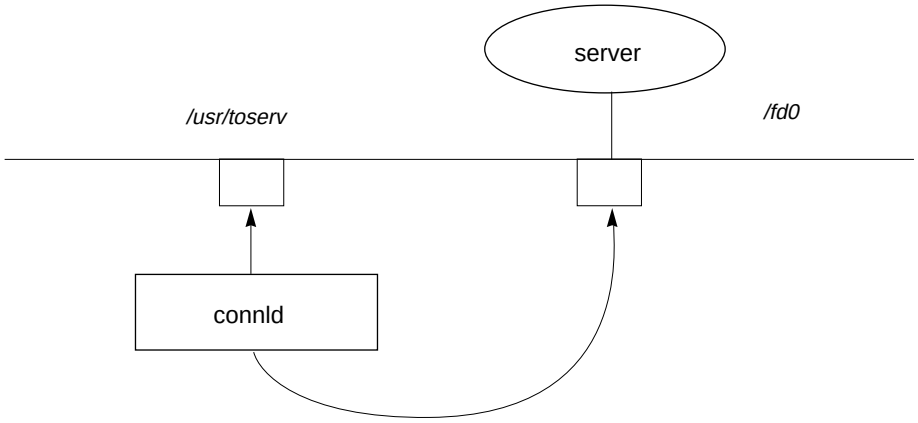


Figure 6–3 Server Sets Up a Pipe

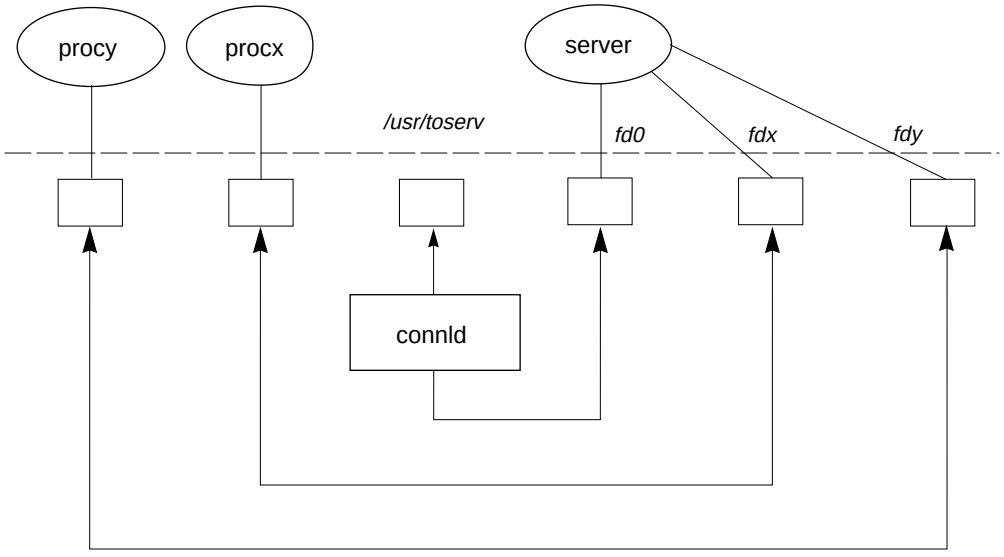


Figure 6–4 Processes X and Y Open /usr/toserv

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.

`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 can not stream the two pipe ends.

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

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

PART II Kernel Interfaces

CHAPTER 7

STREAMS Framework —Kernel Level

The STREAMS subsystem of UNIX provides a framework on which communications services can be built, and as such, 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

Chapter 1describes 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 3discusses messages from the application perspective. The following sections discuss message types, message structure and linkage, how messages are sent and received, and explain 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. A detailed discussion of Message Types is in Chapter 8.

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

TABLE 7–1 Ordinary Messages, Showing Direction of Communication Flow

Ordinary Messages (also called normal messages)		Direction
M_BREAK	Request to a Stream driver to send a “break”	
M_CTL	Control or status request used for intermodule communication	
M_DATA	User data message for I/O system calls	
M_DELAY	Request for a real-time delay on output	
M_IOCTL	Control/status request generated by a Stream head	
M_PASSFP	File pointer-passing message	
M_PROTO	Protocol control information	
M_SETOPTS	Sets options at the Stream head; sends upstream	
M_SIG	Signal sent from a module or driver	

TABLE 7–2 High-Priority Messages, Showing Direction of Communication Flow

High-Priority Messages:		Direction
M_COPYIN	Copies in data for transparent **ioctl**(2)s	
M_COPYOUT	Copies out data for transparent **ioctl**(2)s	
M_ERROR	Reports downstream error condition	
M_FLUSH	Flushes module queue	
M_HANGUP	Sets a Stream head hangup condition	
M_UNHANGUP	Reconnects line, sends upstream when hangup reverses	
M_IOCACK	Positive **ioctl**(2) acknowledgment	
M_IOCDATA	Data for transparent **ioctl**(2)s, sent downstream	
M_IOCNAK	Negative **ioctl**(2) acknowledgment	
M_PCPROTO	Protocol control information	
M_PCSIG	Sends signal from a module or driver	
M_READ	Read notification; sends downstream	
M_START	Restarts stopped device output	
M_STARTI	Restarts stopped device input	

TABLE 7–2 High-Priority Messages, Showing Direction of Communication Flow *(continued)*

High-Priority Messages:		Direction
M_STOP	Suspends output	↓
M_STOPI	Suspends input	↓

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, `db_ref` maintains a reference count of the number of message blocks sharing the data buffer.

Note - `db_base`, `db_lim`, and `db_ref` should not be modified directly. `db_type` are modified under certain conditions, such as changing the message type to reuse the nessage block, provided care is taken.

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` ≥ `b_rptr` ≥≥ `b_wptr` ≥

db_lim. For ordinary messages a priority band can be indicated, and this band is used if the message is queued.

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

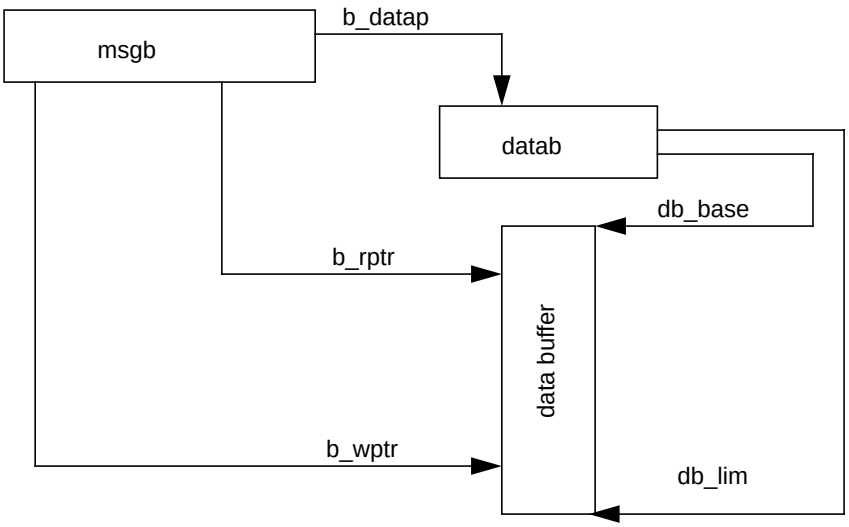


Figure 7–1 Simple Message Referencing the Data Block

The message block has a typedef of mblk_t and has the following public elements:

```
struct msgb {
 struct msgb     *b_next;   /*next msg on queue*/
 struct msgb     *b_prev;   /*previous msg on 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.

Note - 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`.

Note - SunOS has `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.

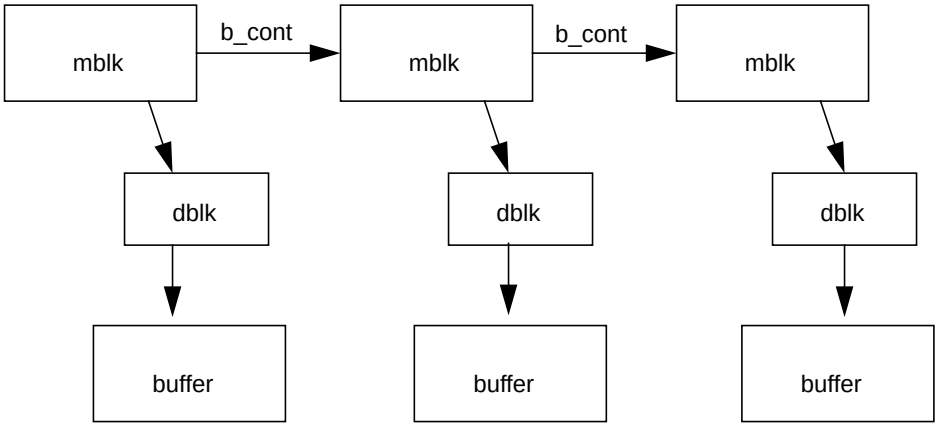


Figure 7–2 Linked Message Blocks

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.

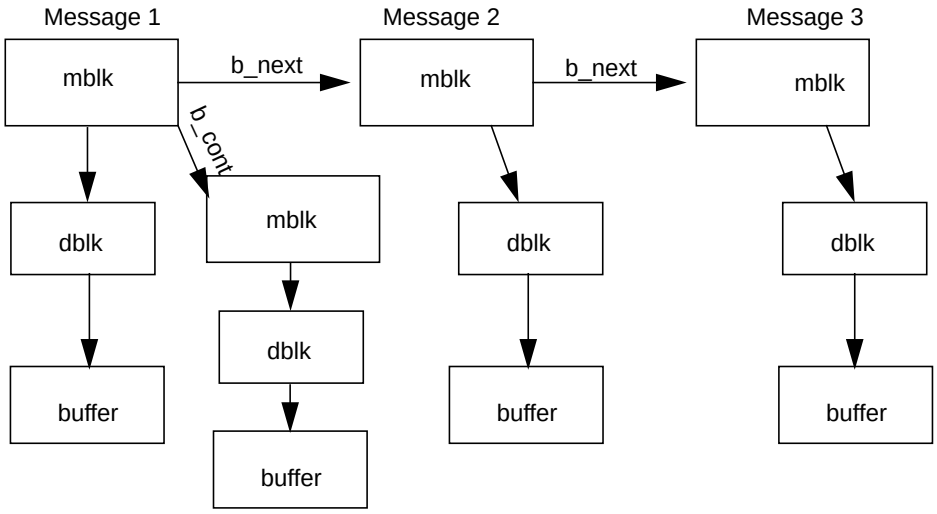


Figure 7–3 Queued Messages

In Figure 7–3messages 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.

Note - 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.

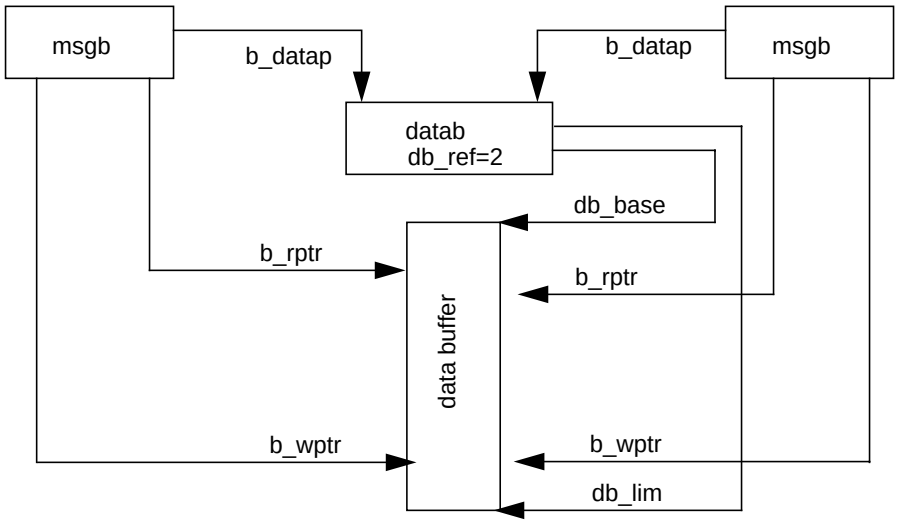


Figure 7–4 Shared Data Block

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, it is recommended that the module copy the message using **copymsg**(9F), free the duplicated message, and then change the appropriate data.

STREAMS provides utility routines and macros (identified in Appendix C “STREAMS Utilities”), to assist in managing messages and message queues, and to assist 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 messages 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 lets modules create messages and pass them to neighboring modules. **read**(2) and **write**(2) are not enough to allow 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 beginning 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.

getmsg(2) and **putmsg**(2) let a user process and the Stream pass data and control information between one another while maintaining distinct message boundaries.

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 schedule,r a module's service procedure processes queued messages in a first-in, first-out (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, after all, 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.

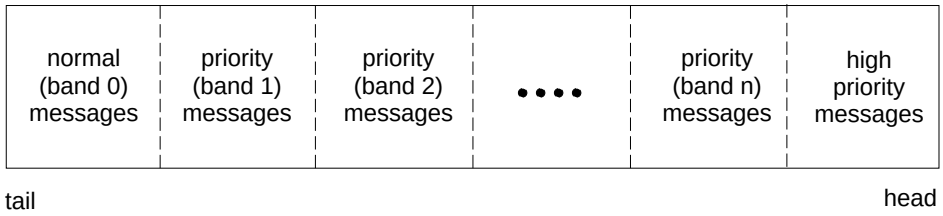


Figure 7–5 Message Ordering in a Queue

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 flow (priority band 1) of data, as shown in Figure 7–6. Queues are explained in detail in the next section.

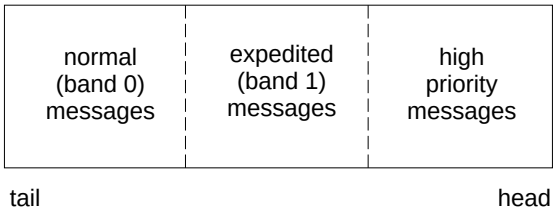


Figure 7–6 Message Ordering with One Priority Band

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, if present. 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.

Note - 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, or an infinite loop results. The enqueuing triggers the queue to be immediately scheduled again.

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. Also 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 Figure Figure 7–7).

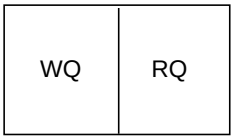


Figure 7–7 Queue Pair Allocation

queue(9S) 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 be to 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 as the `typedef queue_t`, and has the following public elements:

```
struct qinit   *q_qinfo; /* procedure and info for queue */
struct msgb    *q_first; /* head of message queue */
struct msgb    *q_last;  /* tail of message queue */
struct queue   *q_next;  /* next queue in stream */
void      *q_ptr;  /* module private data pointer */
ulong      q_count;  /* number of bytes on queue */
ulong      q_flag;  /* queue state */
long      q_minpsz; /* min packet size accepted by this module */
long      q_maxpsz; /* max packet size accepted by this module */
ulong      q_hiwat;  /* queue high water mark */
ulong      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. See “ **qband**(9S) Structure ” on page 102 for more details. **`qsize`**(9F) can be used to determine the total number of messages on the queue. `q_flag` indicates the state of the queue. See the “Queue Flags” section for the definition of these flags.

`q_minpsz` contains the minimum packet size accepted by the queue, and `q_maxpsz` contains the maximum packet size accepted by this queue. These are suggested limits, and some implementations of STREAMS may not enforce them. The SunOS™ Stream head enforces these values but is voluntary at the module level. 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 allows them to be changed in the queue without modifying the `streamtab` and `module_info` values.

Queue Flags

Be aware of the following queue flags. See **queue**(9S).

TABLE 7–3 Queue Flags

`QENAB`	The queue is enabled to run the service procedure.
`QFULL`	The queue is full.
`QREADR`	Set for all read queues.
`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 C, "STREAMS Utilities"). Modules and drivers should not change any fields not explicitly listed previously.

strqget(9F) lets modules and drivers get information about a queue or particular band of the queue. This insulates the STREAMS data structures from the modules and drivers. The syntax of the routine is shown as:

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

q specifies from which queue the information is to be retrieved; what defines the queue_t field value to obtain (see below). 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) lets modules and drivers change information about a queue or a band of the queue. This also insulates the STREAMS data structures from the modules and drivers. Its format is:

```
int
strqset(queue_t *q. qfields_t what, unsigned char pri, long 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 a pointer. If the field is read-only, the EPERM is returned and the field is left unchanged. The following fields are read-only: QCOUNT, QFIRST, QLAST, and QFLAG.

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.

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`**(9F)).

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 */
    long   mi_minpsz;   /* minimum packet size */
    long   mi_maxpsz;   /* maximum packet size */
    ulong  mi_hiwat;    /* high water mark */
    ulong  mi_lowat;    /* low 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 routine of a device is called once for the initial open of the device and 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. See **open**(9E).

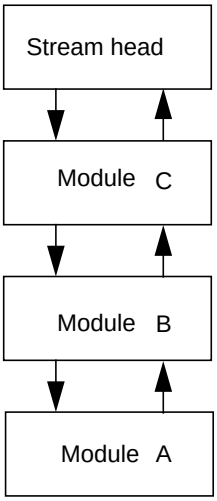


Figure 7–8 Ordering of a Module's open Procedure

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, if a reopen of the Stream occurs, this order reverses. 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 duped (**dup**(2)) the Stream is not reopened.

The syntax of the open entry point is:

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

q points to the read queue of this module. *devp* points 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."

oflag – For devices, *oflag* can contain the following bit mask values: FEXCL, FNDELAY, FREAD, and FWRITE. See Chapter 9 for more information on drivers.

sflag – When the open is associated with a driver, *sflag* is set to 0 or `CLONEOPEN`, see Chapter 9, “Cloning” on page 171 for more details. If the open is associated with a module, *sflag* contains the value `MODOPEN`.

cred_p is a pointer to the user credentials structure.

`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`. So the driver’s interrupt and open routines must allow for this. See Chapter 9 for more information.

`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 recommended.

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`. Code Example 7–1 shows a simple `open` routine.

CODE 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. In other words, messages flow around the module until **`qprocson`**(9F) is called. Figure 7–9illustrates this process.

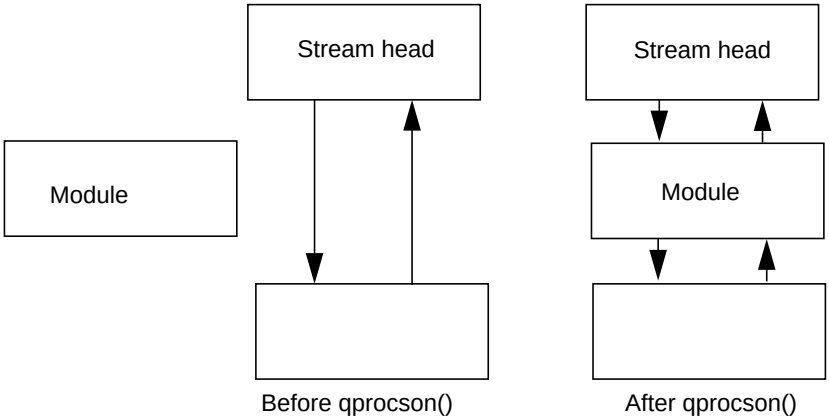


Figure 7–9 Messages Flowing Around the Module Before `qprocson`

The module or driver instance is guaranteed to be single-threaded before `qprocson(9F)` is called except for interrupts or callbacks which must be handled separately. `qprocson(9F)` must be called before calling **`qbufcall`**(9F), **`qtimeout`**(9F), **`qwait`**(9F), or `qwait_sig(9F)`.

`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 syntax of 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. Likewise, the blocking routines should return in the event of a signal. Device drivers must take into consideration that interrupts are not blocked during `close`. The `close` routine must cancel all pending **`timeout`**(9F) and **`qbufcall`**(9F) callbacks, and process any remaining data on its service queue.

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. Code Example 7–2shows an example of a module `close` routine.

```
/* 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);
}
```

The put Procedure

The put procedure passes messages from the queue of a module to the queue of the next module. The queue's put procedure is invoked by the preceding module to process a message immediately (see **put**(9F) and **putnext**(9F)). Almost all modules and drivers must have a put routine. The exception is that the read side driver does not need a put routine because there can be no downstream component to call the put.

A driver's put procedure must do one of:

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

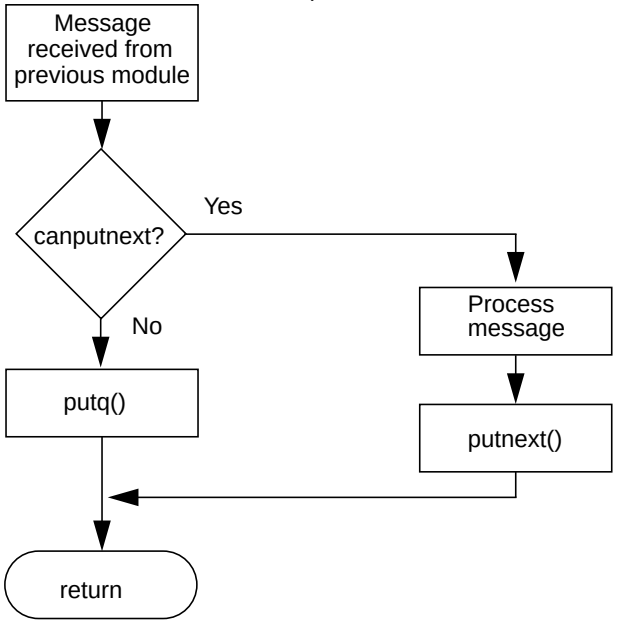
All M_IOCTL type messages must be acknowledged through M_IOACK or rejected through M_IOCNACK. M_IOCTL messages should not be freed. Drivers must free any unrecognized message.

A module's put procedure must do one of:

- 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.

CODE EXAMPLE 7–2 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.

The preferred naming convention for a `put` procedure reflects the direction of the message flow. The read `put` procedure is suffixed by `rput`, and the write procedure by `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, in general, it is better to process them immediately in the `put` procedure. Ordinary or priority-band messages should be placed 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)). Code Example 7–3 shows the put procedure.

CODE EXAMPLE 7–3 A Module's `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-&gt;b_datap-&gt;db_type &gt;= QPCTL ||
              (canputnext(q) &amp;&amp; q-&gt;q_first == NULL)) {
        /*
         * Process message
         */

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

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 has the concept of perimeters (see "MT STREAMS Perimeters" on page 221 in Chapter 12 for information). Perimeters are a facility that lets a module specify that the framework provide exclusive access for the entire module, queue pair, or an individual queue. Perimeters make it easier to deal with multithreaded issues, such as message ordering and recursive locking.

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, don't assume that a module's `put` procedure has been called just because **`put`**(9F), **`putnext`**(9F), or **`qreply`**(9F) have returned.

`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 105 in Chapter 7 for more on Flow Control.)

  The service procedure is optional. Use a service procedure for the following reasons:

- 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 (**`putq`**(9F)) is the first message on the queue,
  - The message being queued (**`putq`**(9F)) is 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 reenabled (**`qenable`**(9F)) at a later time. Figure 7-11 shows the flow of a `service` procedure.

Warning - High-priority messages must never be placed back on a service queue (**`putbq`**(9F)); this can cause an infinite loop.

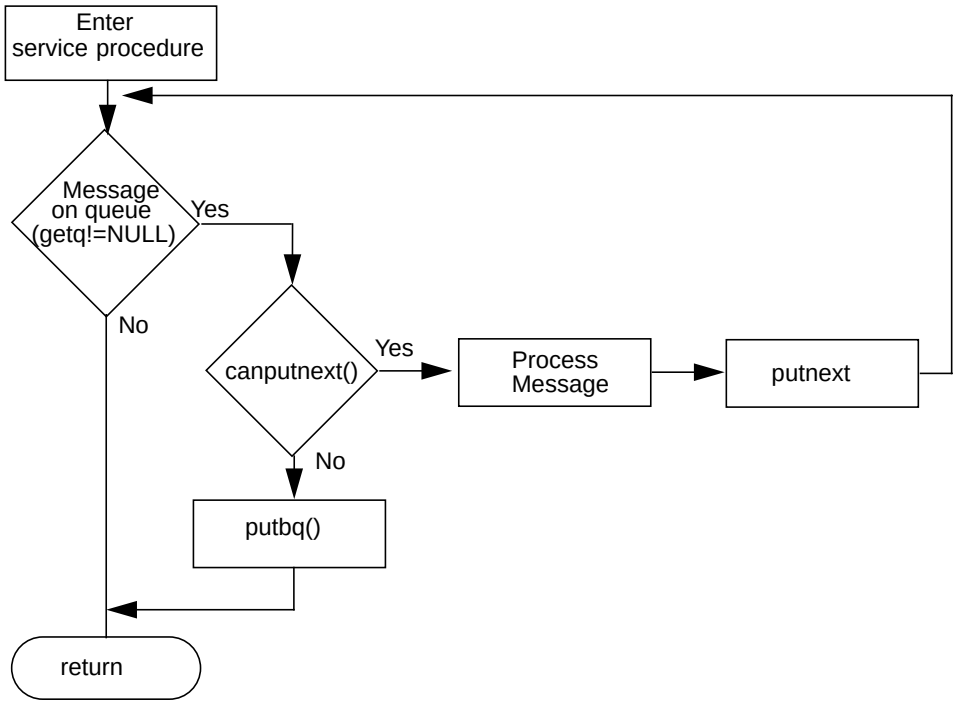


Figure 7–10 Flow of `service` Procedure

Code Example 7–4 shows the module service procedure.

CODE EXAMPLE 7–4 A 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-&gt;b_datap-&gt;db_type &gt;= QPCTL || (canputnext (q)) {
    /*
     * process message
     */
    .
    .
    .
    putnext (q, mp);
   } else {
```

```
    /*
          * put message back on service queue
          */
    putbq(q,mp);
    break;
   }
  }
  return (0);
}
```

qband(9S) Structure

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

```
struct qband *qb_next;  /* next band's info */
 ulong    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 */
 ulong    qb_hiwat;  /* high watermark for band */
 ulong    qb_lowat;  /* low watermark for band */
 ulong    qb_flag;   /* flag, QB_FULL, denotes that a band of*/
           /* data flow is flow controlled */
```

This 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. One flag defined for qb_flag is QB_FULL, which 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**(9S) Information

The STREAMS utility routines should be used when manipulating the fields in the queue and qband 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. There are fields in the structure that are reserved and are not documented.

Figure 7–11shows a queue with two extra bands of flow.

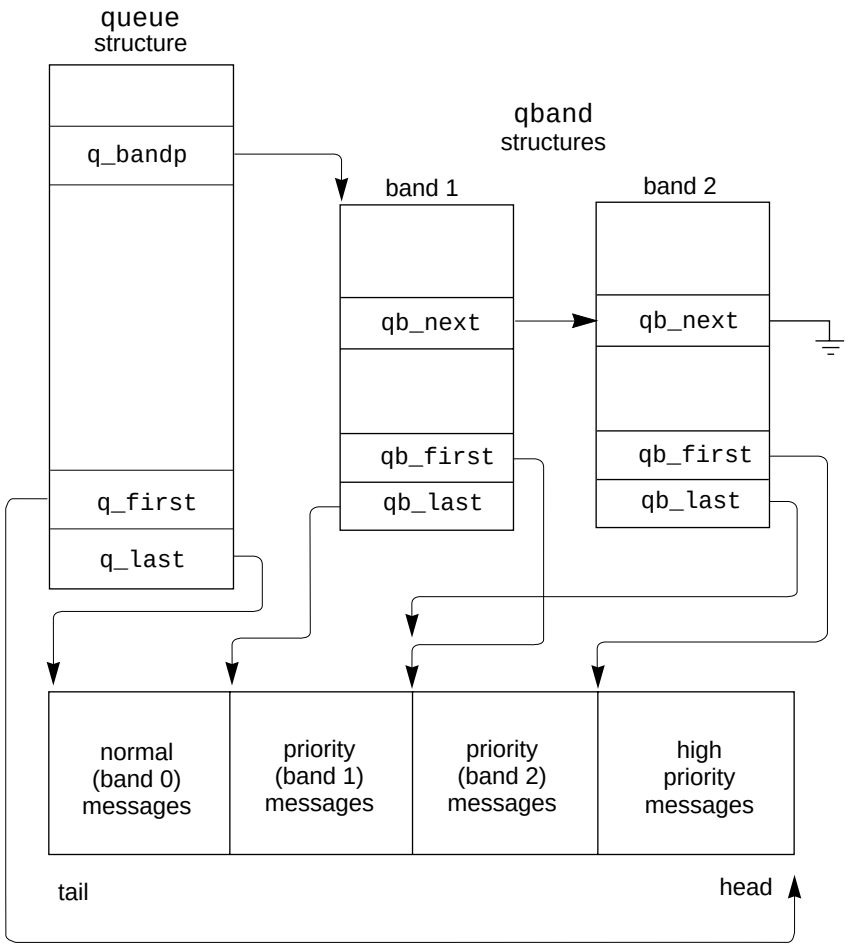


Figure 7–11 Data Structure Linkage

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 “Flush Handling” on page 137. **bcanputnext**(9F) is discussed in , and the other two routines are described in the following section. Appendix B also has a description of these routines.

Message Processing

Typically, put procedures are required in pushable modules, but service procedures are optional. If the put routine queues messages, there must exist a corresponding service routine that handles the queued messages. If the put routine does not queue messages, the service routine need not exist.

Code Example 7–2 shows typical processing flow for a put procedure which works as follows:

- A message is received by the put procedure associated with 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 7–10 shows typical processing flow for a service procedure which 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 by using the **getq**(9F) utility.
- 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 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 would not be calling the module's put procedure. Data would come into the Stream from an interrupt routine, and would either process the message, or queue it for the service procedure.

A STREAMS filter is an example of a module without a service procedure — messages passed to it would either be passed or filtered. Flow control is described in .

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 utilized by the queue. If you don't implement flow control, it is possible to overflow queues 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 7–12). 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 Stream 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.
- 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 Figure 7–12, if D is full and C has no service procedure, then B is the nearest preceding queue.

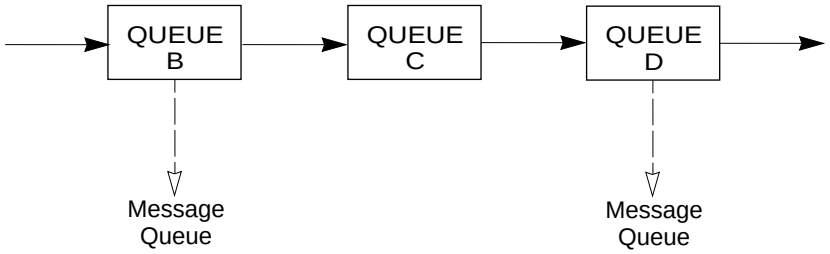


Figure 7–12

- 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 (recall that 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)-&gt;b_datap-&gt;db_type&gt; = 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.

Warning - 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 yet 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), it causes the service procedure to be 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 it must 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(9F) 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.

qband(9S) and **insq**(9F) 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-&gt;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).

“Loop-Around Driver” on page 175 of Chapter 9.

qenable(9F), another flow-control utility, lets a module or driver 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).

CHAPTER 8

Messages - Kernel Level

This chapter describes the structure and use of each STREAMS message type.

ioctl(2) 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 allows 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. If, however, it is 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. 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.

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 routinereturns 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 data.

allocb(9F) can return a buffer larger than the size requested. If **allocb**(9F) indicates 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 Code Example 8–1, **allocb**(9F) is used by the bappend subroutine that appends a character to a message block:

CODE EXAMPLE 8–1 **allocb**(9F) Use

```
/*
 * 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-&gt;b_wptr &gt;= bp-&gt;b_datap-&gt;db_lim)
     return (0);
  } else {
    if ((*bpp = bp = allocb(MODBLKSZ, BPRI_MED)) == NULL)
     return (1);
  }
  *bp-&gt;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, silently discarding the character. If the original message block is not full or the **allocb**(9F) is successful, bappend stores the character in the block.

Code Example 8–2processes all the message blocks in any downstream data (type M_DATA) messages. **freemsg**(9F) frees messages.

CODE EXAMPLE 8–2 Subroutine modwput

```
/* Write side put procedure */
static int modwput(queue_t *q, mblk_t *mp)
{
  switch (mp-&gt;b_datap-&gt;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-&gt;b_cont) {
     while (bp-&gt;b_rptr &lt; bp-&gt;b_wptr) {
       if (*bp-&gt;b_rptr == '\n')
         if (!bappend(&amp;nbp, '\r'))
          goto newblk;
       if (!bappend(&amp;nbp, *bp-&gt;b_rptr))
         goto newblk;
```

```
      bp-&gt;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 a new block(s), 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:

```
int 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 actually still be available.

On success, **bufcall**(9F) 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 will never be called.

Caution - Care must be taken to avoid deadlock when holding resources while waiting for **bufcall**(9F) to call `(*func)(arg)`. **bufcall**(9F) should be used sparingly.

Two examples are provided. Code Example 8–3 is a device-receive-interrupt handler:

CODE EXAMPLE 8–3 Device Interrupt handler

```
#include &lt;sys/types.h&gt;
#include &lt;sys/param.h&gt;
#include &lt;sys/stream.h&gt;
int 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 for more information on the uses of **unbufcall**(9F). These references are protected by MT locks.

Since **bufcall**(9F) can fail, there is still a chance that the device hangs. 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 generally better than hanging.

Code Example 8–4, `mod_wsrv` prefixes each output message with a header.

CODE 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 .... */
   }
  }
 }
```

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. This example deals with **`bufcall`**(9F) failure by calling **`timeout`**(9F). **`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. Since 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, a situation exists where the callback can occur and the driver is closed. This is one of the most difficult types of bugs to track down during the debugging stage.

Warning - All **`bufcall`**(9F)s 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)) that allows the sharing of buffers between users, kernel, and the I/O card.

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 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 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 (for the most part) from the normal data blocks. The client-supplied buffers are processed as if they were normal STREAMS data buffers.

Drivers can not only attach non-STREAMS data buffers but also free them. This is done as follows:

- Allocation - If the drivers use DARAM without using STREAMS resources and without depending on upstream modules, 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 or is not 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 which could free a message block. Otherwise, the possibility for lock recursion and/or deadlock exists.

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), namely, 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, for a detailed description of **esballoc**(9F).

esballoc(9F) Example

Skeletal Code 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 which is allocated by **esballoc**(9F). The allocator free routine uses the queue struct assigned to the driver, or some 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`, the mutex `mp` and the condition variable `cvp` are used to implement the coordination:

CODE EXAMPLE 8–5 esballoc Example

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

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

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

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

  mutex_exit(&amp;mp);
}
```

Warning - Close routine must wait for all **esballoc**(9F) memory to be freed.

General **ioctl**(2) Processing

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, it is the value of the cmd argument in the call to ioctl(2). The ioc_cr field is 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 mblks b_cont field. If the value of the ioc_count field is zero, there will be no M_DATA mblk's 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(2) Processing" on page 56).

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 acknowledgment or M_IOCNAK message upstream. In all circumstances, if a module or driver processes an M_IOCTL message it must acknowledge it.

Modules must always pass unrecognized messages on. Drivers should "nak" 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 the negative acknowledgment 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. The module can set ioc_error to an alternate error number optionally.

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 **ioctl**(2)s 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, 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 the module checks, for example, the TCSETA/TCGETA group of ioctl(2) calls as they pass up or down a Stream, it must never assume that because TCSETA comes down it actually has a data buffer attached to it. The user can have formed TCSETA as an I_STR call and accidentally given a NULL data buffer pointer. One must 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 not there and allocate a new buffer and assign b_cont to point at it, because the original buffer will be lost.

I_STR **ioctl**(2) Processing

Neither the transparent nor nontransparent method implements **ioctl**(2)s in the Stream head, but in the Streams driver or module itself. I_STR ioctl(2)s (also referred to as nontransparent ioctl(2)s) are 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 following:

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

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

ioctl(fd, I_STR, &amp;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)). A 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).

Code Example 8–6, illustrates processing associated with an I_STR **ioctl**(2). lpdoioctl is called by lp's write-side put or service procedure to process M_IOCTL messages:

CODE EXAMPLE 8–6 I_STR **ioctl**(2)

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

  lp = (struct lp *)q-&gt;q_ptr;

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

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

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

lpdoioctl illustrates driver M_IOCTL processing which also applies to modules. In this example, only one command is recognized, SET_OPTIONS. ioc_count contains the number of user-supplied data bytes. For this example, ioc_count must equal the size of a short.

Once the command has been verified (lines 20-24), lpsetopt (not shown here) is called to actually process the request (lines 26-27). lpsetopt returns 0 if the request is satisfied, otherwise an error number is returned.

If ioc_error is nonzero, on receipt of the acknowledgment the Stream head returns -1 to the application's **ioctl**(2) request and set errno to the value of ioc_error.

The **ioctl**(2) is acknowledged (lines 30-33). 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.

This example is for a driver. In the default case, for unrecognized commands, or for malformed requests a nak is generated (lines 34-38). This is done by simply changing the message type to an M_IOCNAK and sending it back up stream. A module does not nak an unrecognized command, but passes the message on. A module does nak a malformed request.

Transparent **ioctl**(2) 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)s also let an application be written using conventional **ioctl**(2) semantics instead of using the I_STR **ioctl**(2) and an strioctl structure. The difference between transparent and nontransparent **ioctl**(2)

ioctl(2) processing in a Streams driver and module is in the way data is transferred from user to kernel space.

The transparent **ioctl**(2) mechanism allow backward compatibility for older programs. This transparency only works for modules and drivers that support transparent **ioctl**(2)s. Trying to use transparent **ioctl**(2)s on a Stream that doesn't 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 allows arbitrary character translations. is pushed on the Stream The **ioctl**(2) specifies the translation to do, and in this case all uppercase vowels are changed to lowercase. A transparent **ioctl**(2) uses XCASE, instead of using I_STR to inform the module directly what should be done.

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

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

Warning - Incorrectly written drivers can cause applications using transparent **ioctl**(2)s to block indefinitely.

Notice that even though this process is simpler in the application, transparent ioctls add 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, it breaks when transparent **ioctl**(2)s are performed with the same command.

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

All M_IOCTL message types (M_COPYIN, M_COPYOUT, M_IOCDATA,M_IOCACK and M_IOCNACK) have some similar data structures and sizes. 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 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 message 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 each 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 (this 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 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 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 completed 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**(2) Examples

Following are three examples of transparent **ioctl**(2) processing. The first illustrates M_COPYIN to copy data from user space. The second illustrates M_COPYOUT to copy data to user space. The third is a more complex example showing state transitions that combine M_COPYIN and M_COPYOUT.

In these examples the message blocks are reused. Some members of the data structures are important and the overhead of allocating, copying, and releasing messages is avoided. 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

Code Example 8–7 illustrates only the processing of a transparent **ioctl**(2) request (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) &amp;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)s the structure (address) and the second **copyin**(9F)s the buffer (address.ad.addr). Two states are maintained and processed in this example: GETSTRUCT is for copying in the address structure, and GETADDR for copying in the ad_addr of the structure.

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). This is done by checking if the size count is equal to TRANSPARENT [line 28). If it is equal to TRANSPARENT, then the message was generated from a transparent **ioctl**(2); that is not from an I_STR **ioctl**(2). [lines 29-32)(

CODE EXAMPLE 8–7 M_COPYIN Example

```
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-&gt;b_datap-&gt;db_type) {
  .
  .
  .
  case M_IOCTL:
   /* Process ioctl commands */
   iocbp = (struct iocblk *)mp-&gt;b_rptr;
   switch (iocbp-&gt;ioc_cmd) {
    case SET_ADDR;
     if (iocbp-&gt;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-&gt;b_rptr;
      /* Get user space structure address from linked
       * M_DATA block */
      cqp-&gt;cq_addr = *(caddr_t *) mp-&gt;b_cont-&gt;b_rptr;
      cqp-&gt;cq_size = sizeof(struct address);
      /* MUST free linked blks */
      freemsg(mp-&gt;b_cont);
      mp-&gt;b_cont = NULL;

      /* identify response */
      cqp-&gt;cq_private = (mblk_t *)GETSTRUCT;

      /* Finish describing M_COPYIN message */
      cqp-&gt;cq_flag = 0;
      mp-&gt;b_datap-&gt;db_type = M_COPYIN;
      mp-&gt;b_wptr = mp-&gt;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-&gt;ioc_cmd) */
   break;
  case M_IOCDATA:
   /* all M_IOCDATA processing done here */
   xxioc(q, mp);
   break;
 }
 return (0);
}
```

The transparent part of the SET_ADDR M_IOCTL message processing requires the address structure to be copied from user address space. To accomplish this, it issues an M_COPYIN request to the Stream head (lines 37-64).

The mblk is reused and mapped into a **copyreq**(9S) structure (line 42. 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) message. (lines 47-49). The b_cont of the copy request mblk is not needed, so it is freed and then NULLed (line 51-52).

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 (line 53). The copy request message is then sent back to the Stream head (line 63). xxwput then returns and is called again when the Stream head responds with an M_IOCDATA message, which is processed by the xxioc routine (lines 72-74).

On receipt of the M_IOCDATA message for the SET_ADDR command, xxioc routine checks cp_rval. If an error occurred during the copyin operation, cp_rval is set. The mblk is freed (lines 93-96) 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.

The cp_private field is set to GETSTRUCT (lines 97-99). This 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 (lines 100-116), 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.

The final M_IOCDATA message arrives from the Stream head. cp_private contains GETADDR(line 118). 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 a M_IOCACK message (lines 124-128). If xx_set_addr fails, the message is rejected with a M_IOCNAK message (lines 121-122). xx_set_addr is a routine (not shown in the example) that 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 zero ioc_error, otherwise an error code could be passed to the user application. ioc_rval and ioc_count are also zeroed to reflect that a return value of 0 and no data is to be passed up (lines 124-128).

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, but is more efficient, as the Stream head frees it.

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.

```
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-&gt;b_rptr;
 iocbp = (struct iocblk *)mp-&gt;b_rptr;

 /* validate this M_IOCDATA is for this module */
 switch (csp-&gt;cp_cmd) {
  case SET_ADDR:
   if (csp-&gt;cp_rval){ /*GETSTRUCT or GETADDRfail*/
    freemsg(mp);
    return;
   }
   switch ((int)csp-&gt;cp_private){ /*determine state*/
    case GETSTRUCT:     /* user structure has arrived */
     /* reuse M_IOCDATA block */
     mp-&gt;b_datap-&gt;db_type = M_COPYIN;
     mp-&gt;b_wptr = mp-&gt;b_rptr + sizeof (struct copyreq);
     cqp = (struct copyreq *)mp-&gt;b_rptr;
     /* user structure */
     ap = (struct address *)mp-&gt;b_cont-&gt;b_rptr;
     /* buffer length */
     cqp-&gt;cq_size = ap-&gt;ad_len;
     /* user space buffer address */
     cqp-&gt;cq_addr = ap-&gt;ad_addr;
     freemsg(mp-&gt;b_cont);
     mp-&gt;b_cont = NULL;
     cqp-&gt;cq_flag = 0;
     cqp-&gt;cp_private=(mblk_t *)GETADDR;  /*nxt st*/
     qreply(q, mp);
     break;

    case GETADDR:      /* user address is here */
     /* hypothetical routine */
     if (xx_set_addr(mp-&gt;b_cont) == FAILURE) {
      mp-&gt;b_datap-&gt;db_type = M_IOCNAK;
      iocbp-&gt;ioc_error = EIO;
     } else {
      mp-&gt;b_datap-&gt;db_type=M_IOCACK;/*success*/
      /* can have been overwritten */
      iocbp-&gt;ioc_error = 0;
      iocbp-&gt;ioc_count = 0;
      iocbp-&gt;ioc_rval = 0;
     }
     mp-&gt;b_wptr=mp-&gt;b_rptr + sizeof (struct ioclk);
     freemsg(mp-&gt;b_cont);
     mp-&gt;b_cont = NULL;
     qreply(q, mp);
```

```
      break;

     default: /* invalid state: can't happen */
      freemsg(mp-&gt;b_cont);
      mp-&gt;b_cont = NULL;
      mp-&gt;b_datap-&gt;db_type = M_IOCNAK;
      mp-&gt;b_wptr = mp-&gt;rptr + sizeof(struct iocblk);
      /* can have been overwritten */
      iocbp-&gt;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;
```

M_COPYOUT Example

Code Example 8–8 returns option values for this Stream 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) &amp;optadd)
```

or by an `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 *)&amp;optadd;
ioctl(fd, I_STR, (caddr_t) &amp;opts_strioctl)
```

In the `I_STR` case, `opts_strioctl.ic_dp` points to the `options` structure, `optadd`.

Code Example 8–8 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 `copyout` the contents of the structure. `xxwput` is the write-side put procedure of module or driver `xx`:

This example first checks if the **`ioctl`**(2) command is transparent (line 22). If it is, the message is reused as an `M_COPYOUT` copy request message (lines 24-32). The pointer to the receiving buffer is in the linked message and is copied into `cq_addr` (lines 26-27). Since 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 (lines 32-33). As an optimization, the following code checks the size of the message for reuse:

A new linked message is allocated to hold the option request. (lines 32-40). When using the transparent **`ioctl`**(2) the `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 a `M_IOCDATA` in response to the `M_COPYOUT` message, which the module must acknowledge in a `M_IOCACK` message (lines 59-73). `ioc_error`, `ioc_count`, and `ioc_rval` are cleared to prevent any stale data from being passed back to the Stream head (lines 69-71).

If the message is not transparent (is issued through an `I_STR` **`ioctl`**(2)) the data is sent with the `M_IOCACK` acknowledgment message and copied into the buffer specified by the `strioctl` data structure (lines 50-51).

CODE EXAMPLE 8–8 `M_COPYOUT` Example

```
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-&gt;b_datap-&gt;db_type) {
  .
  .
  .
  case M_IOCTL:
   iocbp = (struct iocblk *)mp-&gt;b_rptr;
   switch (iocbp-&gt;ioc_cmd) {
    case GET_OPTIONS:
     if (iocbp-&gt;ioc_count == TRANSPARENT) {
      transparent = 1;
      cqp = (struct copyreq *)mp-&gt;b_rptr;
      cqp-&gt;cq_size = sizeof(struct options);
      /* Get struct address from linked M_DATA block */
      cqp-&gt;cq_addr = (caddr_t)
             *(caddr_t *)mp-&gt;b_cont-&gt;b_rptr;
      cqp-&gt;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-&gt;b_cont)
      freemsg(mp-&gt;b_cont);
```

```
   if ((mp-&gt;b_cont =
      allocb(sizeof(struct options), BPRI_MED)) == NULL) {
    mp-&gt;b_datap-&gt;db_type = M_IOCNAK;
    iocbp-&gt;ioc_error = EAGAIN;
    qreply(q, mp);
    break;
   }
   /* hypothetical routine */
   xx_get_options(mp-&gt;b_cont);
   if (transparent) {
    mp-&gt;b_datap-&gt;db_type = M_COPYOUT;
    mp-&gt;b_wptr = mp-&gt;b_rptr + sizeof(struct copyreq);
   } else {
    mp-&gt;b_datap-&gt;db_type = M_IOCACK;
    iocbp-&gt;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-&gt;ioc_cmd) */
 break;

case M_IOCDATA:
 csp = (struct copyresp *)mp-&gt;b_rptr;
 /* M_IOCDATA not for us */
 if (csp-&gt;cmd != GET_OPTIONS) {
  /*if module/pass on, if driver/free message*/
  break;
 }
 if ( csp-&gt;cp_rval ) {
  freemsg(mp); /* failure */
  return (0);
 }
 /* Data successfully copied out, ack */

 /* reuse M_IOCDATA for ack */
 mp-&gt;b_datap-&gt;db_type = M_IOCACK;
 mp-&gt;b_wptr = mp-&gt;b_rptr + sizeof(struct iocblk);
 /* can have been overwritten */
 iocbp-&gt;ioc_error = 0;
 iocbp-&gt;ioc_count = 0;
 iocbp-&gt;ioc_rval = 0;
 qreply(q, mp);
 break;
 .
 .
 .
} /* switch (mp-&gt;b_datap-&gt;db_type) */
return (0);
```

Bidirectional Transfer Example

Code Example 8–9 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) &amp;addr_xxdata)
```

Three pairs of messages are required following the `M_IOCTL` message: the first to copyin the structure; the second to copyin one user buffer; and the third to copyout the second user buffer. `xxwput` is the write-side put procedure for module or driver xx:

CODE EXAMPLE 8–9 Bidirectional 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 GETSTRUCT  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-&gt;b_datap-&gt;db_type) {
   .
   .
   .
   case M_IOCTL:
    iocbp = (struct iocblk *)mp-&gt;b_rptr;
    switch (iocbp-&gt;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-&gt;b_rptr;

    /* Get structure's user address from
     * linked M_DATA block */

    cqp-&gt;cq_addr = (caddr_t)
     *(long *)mp-&gt;b_cont-&gt;b_rptr;
```

```
    freemsg(mp-&gt;b_cont);
    mp-&gt;b_cont = NULL;

    /* Allocate state buffer */

    if ((tmp = allocb(sizeof(struct state),
     BPRI_MED)) == NULL) {
      mp-&gt;b_datap-&gt;db_type = M_IOCNAK;
      iocbp-&gt;ioc_error = EAGAIN;
      qreply(q, mp);
      break;
    }
    tmp-&gt;b_wptr += sizeof(struct state);
    stp = (struct state *)tmp-&gt;b_rptr;
    stp-&gt;st_state = GETSTRUCT;
    cqp-&gt;cq_private = tmp;

    /* Finish describing M_COPYIN message */

    cqp-&gt;cq_size = sizeof(struct xxdata);
    cqp-&gt;cq_flag = 0;
    mp-&gt;b_datap-&gt;db_type = M_COPYIN;
    mp-&gt;b_wptr=mp-&gt;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-&gt;ioc_cmd) */
   break;

 case M_IOCDATA:
   xxioc(q, mp);/*all M_IOCDATA processing here*/
   break;
   .
   .
   .
 } /* switch (mp-&gt;b_datap-&gt;db_type) */
}
```

`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()`:

```
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-&gt;b_rptr;
```

```
 iocbp = (struct iocblk *)mp-&gt;b_rptr;
 switch (csp-&gt;cp_cmd) {

 case XX_IOCTL:
   if (csp-&gt;cp_rval) { /* failure */
    if (csp-&gt;cp_private) /* state structure */
      freemsg(csp-&gt;cp_private);
    freemsg(mp);
    return;
    }
   stp = (struct state *)csp-&gt;cp_private-&gt;b_rptr;
   switch (stp-&gt;st_state) {

   case GETSTRUCT:     /* xxdata structure copied in */
     /* save structure */

    stp-&gt;st_data =
     *(struct xxdata *)mp-&gt;b_cont-&gt;b_rptr;
    freemsg(mp-&gt;b_cont);
    mp-&gt;b_cont = NULL;
    /* Reuse M_IOCDATA to copyin data */
    mp-&gt;b_datap-&gt;db_type = M_COPYIN;
    cqp = (struct copyreq *)mp-&gt;b_rptr;
    cqp-&gt;cq_size = stp-&gt;st_data.x_inlen;
    cqp-&gt;cq_addr = stp-&gt;st_data.x_inaddr;
    cqp-&gt;cq_flag = 0;
    stp-&gt;st_state = GETINDATA; /* next state */
    qreply(q, mp);
    break;

   case GETINDATA: /* data successfully copied in */
    /* Process input, return output */
    if ((mp-&gt;b_cont = xx_indata(mp-&gt;b_cont))
     == NULL) { /* hypothetical */
       /* fail xx_indata */
       mp-&gt;b_datap-&gt;db_type = M_IOCNAK;
       mp-&gt;b_wptr = mp-&gt;b_rptr +
        sizeof(struct iocblk);
      iocbp-&gt;ioc_error = EIO;
      qreply(q, mp);
      break;
    }
    mp-&gt;b_datap-&gt;db_type = M_COPYOUT;
    cqp = (struct copyreq *)mp-&gt;b_rptr;
    cqp-&gt;cq_size = min(msgdsize(mp-&gt;b_cont),
     stp-&gt;st_data.x_outlen);
    cqp-&gt;cq_addr = stp-&gt;st_data.x_outaddr;
    cqp-&gt;cq_flag = 0;
    stp-&gt;st_state = PUTOUTDATA; /* next state */
    qreply(q, mp);
    break;
   case PUTOUTDATA: /* data copied out, ack ioctl */
    freemsg(csp-&gt;cp_private); /*state structure*/
    mp-&gt;b_datap-&gt;db_type = M_IOCACK;
    mp-&gt;b_wtpr=mp-&gt;b_rptr + sizeof (struct iocblk);
c    /* can have been overwritten */
    iocbp-&gt;ioc_error = 0;
    iocbp-&gt;ioc_count = 0;
    iocbp-&gt;ioc_rval = 0;
    qreply(q, mp);
```

```
    break;

   default: /* invalid state: can't happen */
    freemsg(mp-&gt;b_cont);
    mp-&gt;b_cont = NULL;
    mp-&gt;b_datap-&gt;db_type = M_IOCNAK;
    mp-&gt;b_wptr=mp-&gt;b_rptr + sizeof (struct iocblk);
    iocbp-&gt;ioc_error = EINVAL;
    qreply(q, mp);
    break;
   } /* switch (stp-&gt;st_state) */
   break;
 default: /* M_IOCDATA not for us */
   /* if module, pass message on */
   /* if driver, free message */
   break;
 } /* switch (csp-&gt;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 acknowledgment 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.

(Available as I-LIST2 file)

```
#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);
}
```

Flush 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 submiting 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 that can have values described in Table 8–1.

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 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)

ioctl(2) is passed pointer to a bandinfo structure. The bi_pri field indicates the band priority to be flushed (from 0 to 255). The bi_flag field is used to indicate 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;
  int      bi_flag;
};
```

See the section describing M_FLUSHBAND processing in Chapter 8 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. In the figures dotted boxes indicate flushed queues.

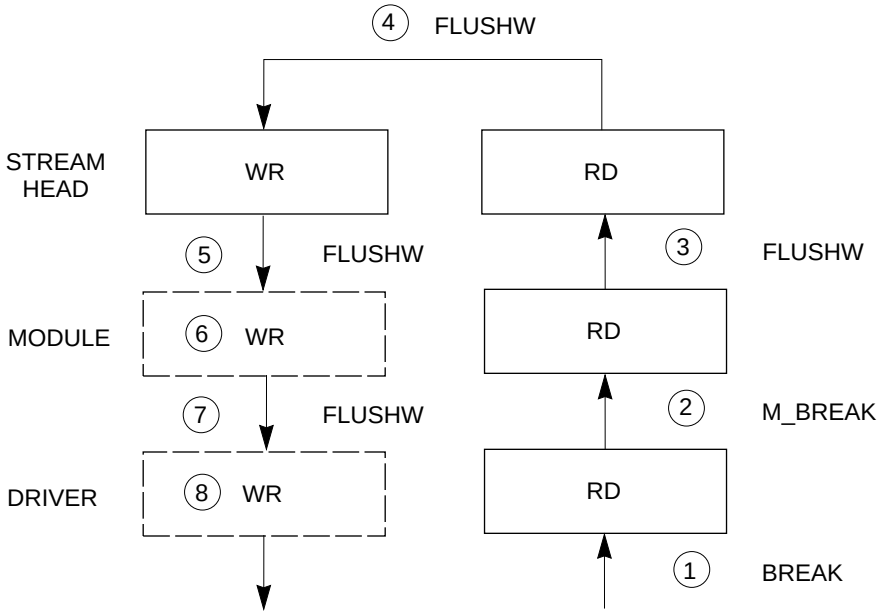


Figure 8–1 Flushing the Write-Side of a Stream

The following takes place (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 and 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 read side of a Stream.

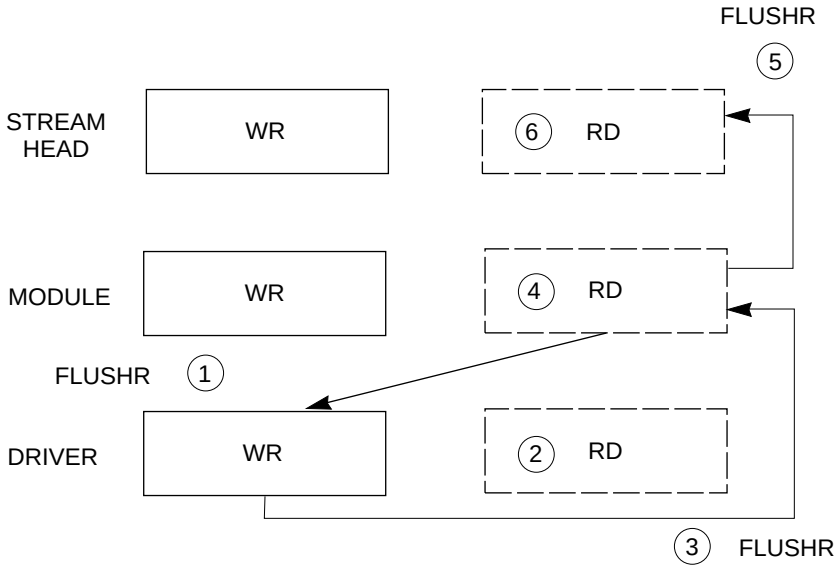


Figure 8–2 Flushing the Read-Side of a Stream

The events taking place are:

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 flush handling:

```
static int
ld_put(
  queue_t *q,      /* pointer to read/write queue */
  mlkb_t *mp)      /* pointer to message being passed */
{
  switch (mp-&gt;b_datap-&gt;db_type) {
   default:
    putq(q, mp); /* queue everything */
   return (0);      /* except flush */

   case M_FLUSH:
    if (*mp-&gt;b_rptr &amp; FLUSHW)     /* flush write q */
      flushq(WR(q), FLUSHDATA);

    if (*mp-&gt;b_rptr &amp; FLUSHR)     /* flush read q */
      flushq(RD(q), FLUSHDATA);

    putnext(q, mp);           /* pass it on */
```

```
        return(0);
      }
    }
```

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:

```
queue_t *rdq;        /* read queue */
queue_t *wrq;        /* write queue */

 case M_FLUSH:
  if (*bp-&gt;b_rptr &amp; FLUSHBAND) {
   if (*bp-&gt;b_rptr &amp; FLUSHW)
    flushband(wrq, FLUSHDATA, *(bp-&gt;b_rptr + 1));
   if (*bp-&gt;b_rptr &amp; FLUSHR)
    flushband(rdq, FLUSHDATA, *(bp-&gt;b_rptr + 1));
  } else {
   if (*bp-&gt;b_rptr &amp; FLUSHW)
    flushq(wrq, FLUSHDATA);
   if (*bp-&gt;b_rptr &amp; 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 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.

One use of the field b_flag of the msgb structure is provided to give the Stream head a way 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.

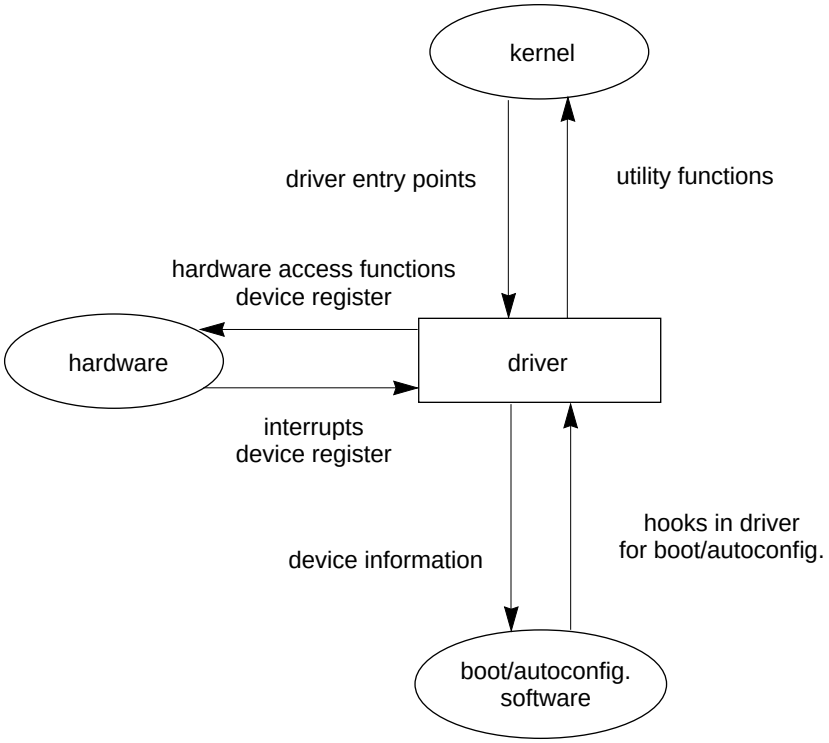


Figure 8–3 Interfaces Affecting Drivers

The set of STREAMS utilities available to drivers are listed in Appendix B. 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 Solaris system releases are implemented as *functions* in the SunOS 5.x System. This does not preclude the existence of both *macro* and *function* versions of these utilities. It is intended that driver source code include a header file that picks up *function* declarations while the core operating system source includes 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 functions **`drv_getparm`**(9F) fetches information from these structures. This restriction has an important consequence. Since 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 defined at the boundary between a service user and a service provider (see Figure 8–4). A service interface is a set of primitives and the rules that 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, with the second through last blocks 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. (Although its use is not recommended, upstream PROTO messages can have multiple PROTO blocks at the start of the message. **getmsg**(2) compacts the blocks into a single control part when sending to a user process.) The M_DATA block(s) 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 respectively 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 is normally used to acknowledge primitives composed of other messages. M_PCPROTO ensures that the acknowledgment 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).

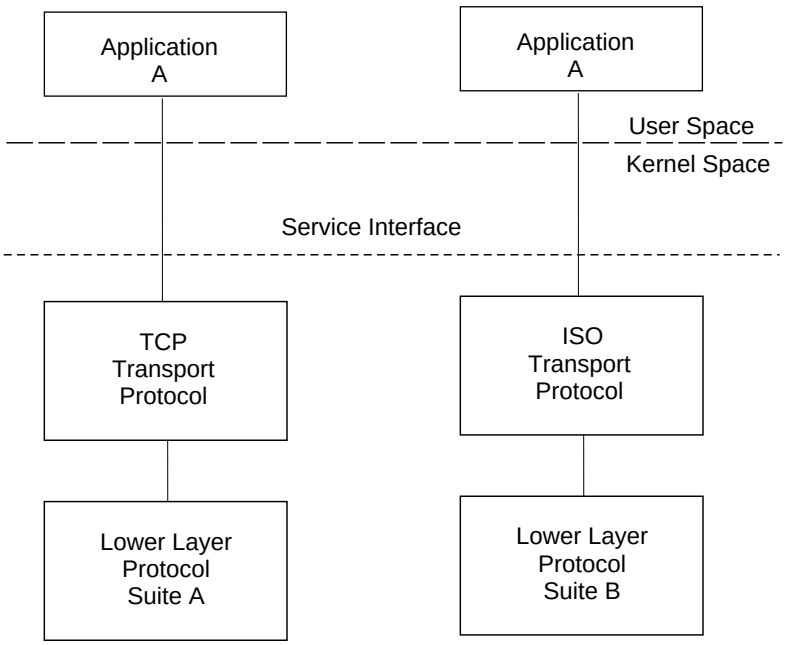


Figure 8–4 Protocol Substitution

By defining a service interface through which applications interact with a transport protocol, you can substitute a different protocol below the service interface completely transparently to the application. In Figure 8–5, 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.

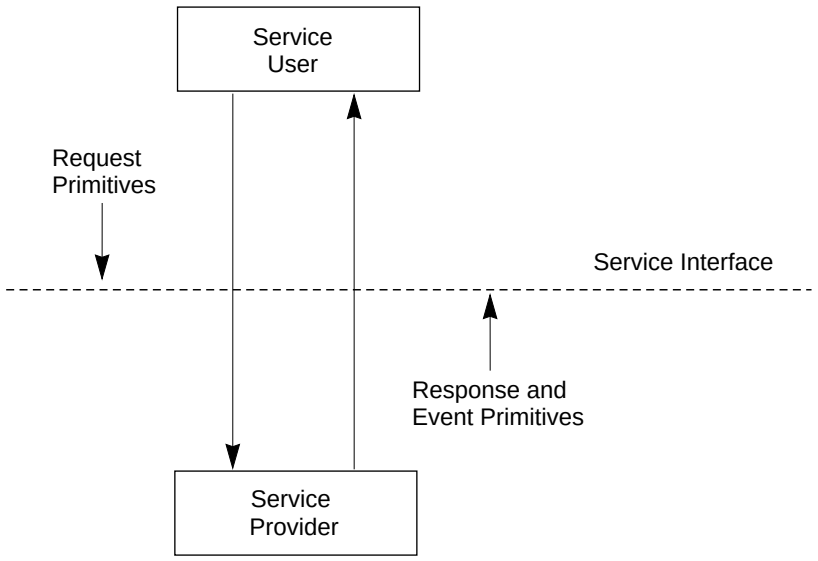


Figure 8–5 Service Interface

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 would be used to define this primitive. The control part would identify the primitive as a connect request and would include the protocol address and options. The data part would contain the associated user data.

Service Interface Library Example

The service interface library example presented here includes four functions that let 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.

```
/*
 * 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 acknowledgment */
 #define ERROR_ACK       4 /* error acknowledgment */
 #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 */
  long   PRIM_type;      /* always BIND_REQ */
  long   BIND_addr;      /* addr to bind */
};
 struct unitdata_req {       /* unitdata request */
  long   PRIM_type;      /* always UNITDATA_REQ */
  long   DEST_addr;      /* destination addr */
};

struct ok_ack {         /* positiv acknowledgment*/
  long   PRIM_type;      /* always OK_ACK */
};

struct error_ack {        /* error acknowledgment */
  long    PRIM_type;      /* always ERROR_ACK */
  long   UNIX_error;      /* UNIX systemerror code */
};

struct unitdata_ind {       /* unitdata indication */
  long   PRIM_type;      /* always UNITDATA_IND */
  long   SRC_addr;      /* source addr */
};

/* union of all primitives */
union 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;
};

/* header files needed by library */
```

```
#include &lt;stropts.h&gt;
#include &lt;stdio.h&gt;
#include &lt;errno.h&gt;
```

Five primitives are defined. The first two represent requests from the service user to the service provider. These are:

`BIND_REQ`	This request asks the provider to bind a specified protocol address. It requires an acknowledgment from the provider to verify that the contents of the request were syntactically correct.
`UNITDATA_REQ`	This request asks the provider to send data to the specified destination address. It does not require an acknowledgment from the provider.

The three other primitives represent acknowledgments of requests, or indications of incoming events, and are passed from the service provider to the service user.

`OK_ACK`	This primitive informs the user that a previous bind request was received successfully by the service provider.
`ERROR_ACK`	This primitive 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`	This primitive indicates that data destined for the user have arrived.

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.

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.

Declarations

The service interface primitives are defined in the declarations:

```
#include &lt;sys/types.h&gt;
#include &lt;sys/param.h&gt;
#include &lt;sys/stream.h&gt;
#include &lt;sys/errno.h&gt;

/* 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 acknowledgment */
#define ERROR_ACK        4 /* error acknowledgment */
#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 */
  long   PRIM_type;      /* always BIND_REQ */
  long   BIND_addr;      /* addr to bind */
};
struct unitdata_req {       /* unitdata request */
  long   PRIM_type;      /* always UNITDATA_REQ */
  long   DEST_addr;      /* dest addr */
};
struct ok_ack {         /* ok acknowledgment */
  long   PRIM_type;      /* always OK_ACK */
};
struct error_ack {        /* error acknowledgment */
  long   PRIM_type;      /* always ERROR_ACK */
  long   UNIX_error;      /* UNIX system error code*/
};
struct unitdata_ind {       /* unitdata indication */
  long   PRIM_type;      /* always UNITDATA_IND */
  long   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 is that structure.

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 acknowledgment (ack) of BIND_REQ.
ERROR_ACK	A negative acknowledgment (nak) of BIND_REQ.
UNITDATA_IND	Data from the network have been received.

The acknowledgment 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 acknowledgment reaches the user before any other message. For example, a UNITDATA_IND could come through before the bind has completed, and the application would be confused.

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

It is assumed (though not shown) that the module open procedure sets the write queue q_ptr to point at the appropriate private data structure.

The write put procedure is:

```
static int protowput(queue_t *q, mblk_t *mp)
{
  union primitives *proto;
  struct dgproto *dgproto;
  int err;
  dgproto = (struct dgproto *) q-&gt;q_ptr;  /* priv data struct */
  switch (mp-&gt;b_datap-&gt;db_type) {
  default:
    /* don't understand it */
    mp-&gt;b_datap-&gt;db_type = M_ERROR;
    mp-&gt;b_rptr=mp-&gt;b_wptr=mp-&gt;b_datap-&gt;db_base;
    *mp-&gt;b_wptr++ = EPROTO;
    qreply(q, mp);
    break;
  case M_FLUSH: /* standard flush handling goes here ... */
    break;
  case M_PROTO:
    /* Protocol message -&gt; user request */
    proto = (union primitives *) mp-&gt;b_rptr;
    switch (proto-&gt;type) {
    default:
     mp-&gt;b_datap-&gt;db_type = M_ERROR;
     mp-&gt;b_rptr=mp-&gt;b_wptr=mp-&gt;b_datap-&gt;db_base;
     *mp-&gt;b_wptr++ = EPROTO;
     qreply(q, mp);
     return;
    case BIND_REQ:
     if (dgproto-&gt;state != IDLE) {
       err = EINVAL;
       goto error_ack;
     }
     if (mp-&gt;b_wptr - mp-&gt;b_rptr !=
      sizeof(struct bind_req)) {
       err = EINVAL;
       goto error_ack;
     }
     if (err = chkaddr(proto-&gt;bind_req.BIND_addr))
       goto error_ack;
     dgproto-&gt;state = BOUND;
     dgproto-&gt;addr = proto-&gt;bind_req.BIND_addr;
     mp-&gt;b_datap-&gt;db_type = M_PCPROTO;
     proto-&gt;type = OK_ACK;
     mp-&gt;b_wptr=mp-&gt;b_rptr+sizeof(structok_ack);
     qreply(q, mp);
     break;
   error_ack:
     mp-&gt;b_datap-&gt;db_type = M_PCPROTO;
     proto-&gt;type = ERROR_ACK;
     proto-&gt;error_ack.UNIX_error = err;
     mp-&gt;b_wptr=mp-&gt;b_rptr+sizeof(structerror_ack);
     qreply(q, mp);
     break;
    case UNITDATA_REQ:
     if (dgproto-&gt;state != BOUND)
       goto bad;
     if (mp-&gt;b_wptr - mp-&gt;b_rptr !=
```

```
      sizeof(struct unitdata_req))
      goto bad;
    if(err=chkaddr(proto-&gt;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 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 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 simply 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 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 sent upstream.

Message Type Change Rules

- You can only change M_IOCTL family of message types to other M_IOCTL message types.
- M_DATA, M_PROTO, M_PCPROTO are dependent on the modules, drivers and service providers 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.

Signals

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`**(5)), 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

STREAMS Drivers

This chapter describes the operation of STREAMS drivers and some of the processing typically required in them.

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* applies to STREAMS drivers as well. For more information on global driver issues and non-STREAMS drivers, see *Writing Device Drivers*.

STREAMS drivers share a basic programming model with STREAMS modules; information common to both drivers and modules is discussed in Chapter 10. 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 Topics

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 Solaris system, 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. Drivers that adhere to the specified interfaces are forward compatible with future releases of the Solaris system.

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, such 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, such 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, such 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 handling interrupts. In STREAMS, drivers are *opened* and modules are *pushed*. A device driver has an interrupt routine to process hardware interrupts.

STREAMS Driver Topics

STREAMS drivers have five different points of contact with the kernel:

Configuration (kernel dynamic loading) entry points. These are the routines that allow 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 — The entry points allow 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), **identify**(9E), **probe**(9E), **attach**(9E), and **detach**(9E).

Table-driven entry points — The table-driven entry points are 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, 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 — These entry points are contained in the `streamtab`; they 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 — A driver's interrupt routine handles 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).

Each of these points of contact is discussed in the following sections.

STREAMS Configuration Entry Points

As with other SunOS 5.x 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. See the appropriate chapters in *Writing Device Drivers* (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.).

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

STREAMS drivers' initialization entry points must perform the same tasks as those of non-STREAMS drivers. See *Writing Device Drivers* 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 9–1shows 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.

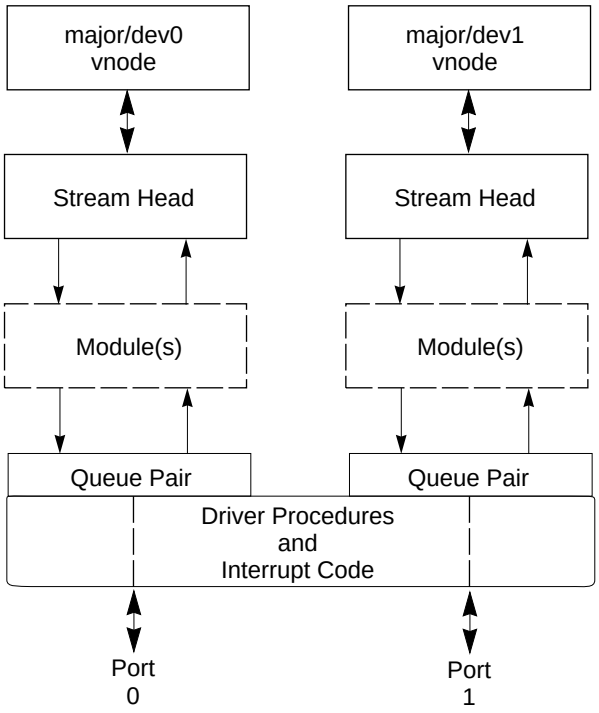


Figure 9–1 Device Driver Streams

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 "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, allowing 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 93.

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 simply 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 more information.

Driver Unloading

STREAMS drivers can prevent unloading through the standard driver **detach**(9E) entry point.

STREAMS Driver Sample Code

The following code samples illustrate three STREAMS driver topics:

Basic hardware/pseudo drivers — This type of driver communicates with a specific piece of hardware (or simulated hardware). The `lp` example that followssimulates 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 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” on page 171 for more information.

Printer Driver Example

The first example is a simple interrupt-per-character line printer driver. 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, unlike a module, 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 would presumably 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 message the printer is operating on), `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 DDK provided "soft-state" functions 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).

CODE 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
};
```

Code Example 9–2shows 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.

CODE EXAMPLE 9–2

```
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));
}
```

Code Example 9–3 shows the lp driver's implementation of the initialization entry points. In `lpidentify`, the driver simply ensures that the name of 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.

CODE EXAMPLE 9–3

```
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;
 register 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 Code 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 data structures are released.

CODE EXAMPLE 9–4

```
/*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 Code 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 it is not necessary to flush it. 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`. Messages types that are not recognized are discarded (in the default case of the switch).

CODE EXAMPLE 9–5

```
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);
}
```

Driver Interrupt

Code 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. It is assumed that this 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, real hardware would generate a hard interrupt, so the call to `ddi_trigger_softintr(9F)` would be unnecessary.

CODE EXAMPLE 9–6

```
/* 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") 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 allows flow control to 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") to modify the Stream head read-side flow control limits.

Cloning

In previous examples, each user process connects a Stream to a driver by explicitly opening a particular minor device of the driver. Each minor device has 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. 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.

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 Code 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 Code 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 Code 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.)

CODE EXAMPLE 9–7

```
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(
```

```
 register 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 */
{
 register struct pt_ttys *ptmp;
 register mblk_t  *mop; /* ptr to a setopts message block */
 register 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 Code 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)'s two interfaces: the first write-only, the second read-write (see the manual 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`

CODE EXAMPLE 9–8

```
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 = (caddr_t)lp;
 WR(q)->q_ptr = (caddr_t)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.

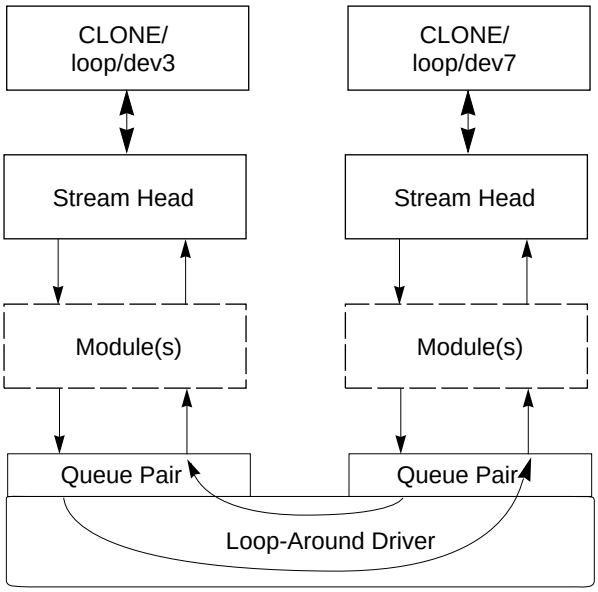


Figure 9–2 Loop-Around Streams

The next example 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 `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" presents multiprocessor and multithreading issues such as locking for data corruption and to prevent race conditions.

Code Example 9–9contains the declarations for the driver.

CODE EXAMPLE 9–9

```
/* 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));
}
```

Code Example 9–10contains the initialization routines.

CODE EXAMPLE 9–10

```
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)
{
 register 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 Code 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.

CODE EXAMPLE 9–11

```
/*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);
}
```

Since 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` (Code 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 sanity 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. The 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.

CODE EXAMPLE 9–12

```
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 then 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 Code 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 can not 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.

CODE EXAMPLE 9–13

```
staticintloopwsrv(queue_t*q)

{
 mblk_t*mp;
 structloop*loop;
 loop=(structloop*)q->q_ptr;

 while((mp=getq(q))!=NULL){
  /*Checkifwecanputthemessageup
  *theotherStreamreadqueue
  */

  if(mp->b_datap->db_type<=QPCTL&&!canputnext(loop->oqptr)){
   (void)putbq(q,mp);/*read-sideisblocked*/
   break;
  }

  /*
  *sendmessagetoqueuefollowing
  *otherStreamreadqueue
  */

  (void)putnext(loop->oqptr,mp);

 }
 return(0);
}

staticintlooprsrv(queue_t*q)
{

 /*Enteronlywhen''backenabled''byflowcontrol*/
 structloop*loop;

 loop=(structloop*)q->q_ptr;

 if(loop->oqptr==NULL)
  return(0);
```

```
 /*manuallyenablewriteserviceprocedure*/
 qenable(WR(loop->oqptr));

 return(0);
}
```

`loopclose` breaks the connection between the Streams, as shown in Code Example 9–14. `loopclose` sends an `M_HANGUP` message up the connected Stream to the Stream head.

CODE EXAMPLE 9–14

```
/*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).

Summary

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), **probe**(9E), and **identify**(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.

Also see *Writing Device Drivers.*

Answers to Frequently Asked Questions

Solaris 2.x Ethernet drivers, **le**(7D) and **ie**(7D) both support Data Link Provider Interfaces (DLPI).

When an `ifconfig` device0 plumb is issued the driver immediately receives a `DL_INFO_REQ`. The information requested by `DL_INFO_ACK` is shown in the `dl_info_ack_t` struct in `/usr/include/sys/dlpi.h`.

A driver can be a CLONE driver and also a DLPI Style 2 provider. Mapping minor numbers selected in the the open routine to an instance prior to a DL_ATTACH_REQ using the instance in the getinfo routine is not valid prior to the DL_ATTACH_REQ. The DL_ATTACH_REQ request is to assign a physical point of attachment (PPA) to a Stream. The DL_ATTACH_REQ request can be issued any time after a file or Stream is opened. The DL_ATTACH_REQ request is not involved in assigning, retrieving, or mapping minor or instance numbers. You can issue a DL_ATTACH_REQ request for a file or Stream with a desired major/minor number. Mapping minor number to instance reflects, in most cases, that the minor number (getmino(dev)) is the instance number.

Each time a driver's attach routine is called, a minor node is created. If a non-CLONE driver needs to attach to multiple boards, that is, to have multiple instances and still create only one minor node, it is possible to use the bits of information in a particular minor number; for example 'FF' to map to all other minor nodes.

CHAPTER 10

Modules

This chapter provides specific examples of how modules work, based on 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 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 complete before running user-level code. The put and service procedures must not block because there is no thread synchronization being done.

Code Example 10–1 shows a STREAMS module read-side put procedure.

CODE 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 switches 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, but not required, by the `put` procedure and passed to the next module.
- Ordinary (or normal) messages are either queued or passed along the Stream.

Code Example 10–2 shows a module write-side `put` procedure.

CODE 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 */
   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 or */
     /* pass message along with putnext(q,mp) */
   return (0);

 case M_PROTO:
c   .
   .
   .

 case M_IOCTL:       /* if cmd in msg is recognized */
        /* process message and send back
reply */
        /* 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. It is up to the module to recognize and process the **`ioctl`**(2) command, or pass the message downstream if it does not recognize the command.

Code Example 10–3 shows a general scenario employed by the module's `service` procedure.

CODE EXAMPLE 10–3 Service Procedure

```
static int
modrsrv(queue_t *q)
{
   mblk_t *mp;

   while ((mp = getq(q)) != NULL) {
         if (!(mp->b_datap->db_type >= QPCTL) &&
            !canputnext(q)) {       /* flow control check *
            putbq(q, mp);      /* return message */
            return (0);
         }
         /* process the message */
         switch(mp->b_datap->db_type) {
            .
            .
            .
            putnext(q, mp); /* pass the result */
         }
      return (0);
}
```

The steps are:

- Retrieve the first message from the queue using **`getq`**(9F).
- 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 watermark, **`canputnext`**(9F) returns false (zero). If the next queue with a `service` procedure is within acceptable flow-control limits, **`canputnext`**(9F) returns true (nonzero).
- 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.
- 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.

- 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 Code 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.

CODE EXAMPLE 10–4 `crmod`

```
/* 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.

Note that, 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 and low watermarks (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 Code Example 10–5.

CODE EXAMPLE 10–5

```
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;
```

modwput, the write put procedure, switches on the message type. 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.

modwsrv is the write service procedure. It takes a single argument, a pointer to the write queue. modwsrv processes only one high-priority message, M_FLUSH. No other high priority-messages should reach modwsrv.

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 module specific.

Ordinary messages are returned to the queue if **canputnext**(9F) returns false, indicating the downstream path is blocked. The example continues with the remainder of `modwsrv` processing `M_DATA` messages:

```
  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);
    }
   }
  }
 }
 return (0);
}
```

The differences in `M_DATA` processing between this and the example in “Message Allocation and Freeing” on page 112 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 cannot be 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 done 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.

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 and low watermarks. 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. The limits prevent depletion of buffers in the buffer pool. Flow control is advisory in nature and can be bypassed. It is managed by high and low watermarks 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 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 modules 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 two examples show a line discipline module's flow control. Code Example 10–6 is a read-side line discipline module and the second shows a write-side line discipline module. Note that the read side is the same as the write side but without the `M_IOCTL` processing.

CODE 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 does not understand the message types, the message types must be passed to the next module.
- The module that acts on an M_IOCTL message sends an M_IOCACK or M_IOCNAK message in response to the **ioctl**(2). If the module does not understand the **ioctl**(2), it passes the M_IOCTL message to the next module.
- Design modules so that they don't pertain to any particular driver but can be used 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 makes it easier to arbitrarily push modules 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 do 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.

Answers to Frequently Asked Questions

TCP and IP are STREAMS modules in the Solaris 2.x system. The command “strconf < /dev/tcp shows you all the modules. STREAMS is not supported in SunOS 4.x system TCP/IP.

Solaris 2.x system DLPI provides both connection-oriented and connectionless services, and multicast features. See **dlpi**(7P).

IP multicast is a standard supported feature in the Solaris 2.x system. In the SunOS 4.x system multicasting is not supported. But, it is available using anonymous ftp from `gregorio.stanford.edu` in the file `vmtp-ip/ipmulti-sunos41x.tar.Z`.

IP is a STREAMS module in the Solaris 2.x system, and 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.

CHAPTER 11

Configuration

This chapter contains information about configuring STREAMS drivers and modules into the Solaris 2.x system. 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*.

This chapter also includes a list of STREAMS-related tunable parameters and describes the **autopush**(1M) facility.

Configuring STREAMS Drivers and Modules

The following sections contain descriptions of the pointer relationships maintained by the kernel and the various data structures used in STREAMS drivers. For the kernel to access a driver, it uses a sequence of pointers in various data structures. Look 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 structure traversal 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 info about the driver being loaded, 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 identify, 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: 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 is a pointer named `streamtab`. If the driver is not a STREAMS driver, `streamtab` is NULL. If the driver is a STREAMS driver, `streamtab` points to a structure that contains initialization routines for the driver.

modlinkage(9S)

This is the definition of **modlinkage**(9S).

```
struct modlinkage {
  int  ml_rev;    /* rev of loadable modules system */
  void  *ml_linkage[4]; /* NULL terminated list of linkage
             * structures */
};
```

modldrv(9S)

The definition of **modldrv**(9S) is.

```
struct modldrv {
  struct mod_ops  *drv_modops;
  char     *drv_linkinfo;
  struct dev_ops  *drv_dev_ops;
};
```

modlstrmod(9S)

This is the definition of **modlstrmod**(9S). 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(9S)

The first structure is **dev_ops**(9S). It 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. Each **dev_ops**(9S) structure contains a **cb_ops**(9S).

```
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(9S)

The **cb_ops**(9S) structure is the SunOS 5.x 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(9S)

The **streamtab**(9S) structure contains pointers to the structures that hold the routines that actually initialize the reading and writing for module.

If streamtab is NULL, it signifies no STREAMS routines and the entire driver is treated as though it were a regular 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(9S)

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)*/
};
```

Entry Points

As described in Chapter 9, and as seen in the previous data structures, there are four entry points:

1. Kernel module loading - `_init(9E)`, **`_fini`**(9E), `_info(9E)`
2. `dev_ops` - **`identify`**(9E), **`attach`**(9E), **`getinfo`**(9E)).
3. `cb_ops` - **`open`**(9E), **`close`**(9E), **`read`**(9E), **`write`**(9E), **`ioctl`**(9E).
4. `streamtab` - **`put`**(9E), **`srv`**(9E).

`pts`(7D) example

Now look at a real example taken from the Solaris 2.x system. The driver **`pts`**(7D) is the pseudo terminal slave driver.

```
                            /*
 * 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_tstatic 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

Here are the structures 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).

```
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

Here are some compile, assemble, and link lines for an example driver with two C modules and an assembly language module.

```
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 more information on the sequence of installing and loading device drivers. The procedure is copy your driver or module to `/kernel/drv` or `/kernel/strmod` respectively, and for drivers run **`add_drv`**(1M).

Checking Module Type

Next, see the 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 refered to by STREAMS are configurable when building a new operating system (see the file `/etc/system` and the SunOS User's Guide to System Administration 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 effectively 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 exceeding this size fails with `ERANGE`.

`autopush`(1M) Facility

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 **`sad`**(7D)) provides an interface to the autopush mechanism. System administrators can open the SAD driver and set or get **`autopush`**(1M) 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 and not been 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.

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 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 **`sad`**(7D) for more information.

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

The 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:

```
/*
 * 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 */
  long apc_major;     /* major device number */
  long apc_minor;     /* minor device number */
  long 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 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:

```
# /dev/console and /dev/contty autopush setup
#
# 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 and 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.

CHAPTER 12

MultiThreaded STREAMS

This chapter describes how to multithread a 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. Refer also to *Writing Device Drivers.*

MT STREAMS Overview

The SunOS 5.x operating system is fully multithreaded, able to 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 new concepts and terminology. These concepts apply not only to STREAMS drivers, but to all device drivers in the SunOS 5.x system. For more a complete description of these terms, see *Writing Device Drivers*. Additionally, see Chapter 1 of this guide for definitions, and Chapter 8 for elements of MT drivers.

You need to understand the following terms and ideas.

Thread	a sequence of instructions executed within context of a process
Lock	a mechanism to restrict access to data structures
Single Threaded	restricting access to a single thread
Multithreaded	allowing two or more threads access

Multiprocessing	two or more CPUs concurrently executing the OS
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 specific event, call module function

MT STREAMS Framework

The STREAMS framework consists of the Stream head, STREAMS utility routines, and documented STREAMS data structures. The STREAMS framework allows multiple kernel threads to concurrently enter and execute within each module. Multiple threads can be actively executing in the `open`, `close`, `put`, and `service` procedures of each queue within the system.

A goal of the SunOS 5.x system is to preserve the interface and flavor of STREAMS to shield module code as much as possible from the impact of migrating to the multithreaded kernel. The majority 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`), it does not have to explicitly lock these framework data structures.

A second goal is to make it simple to write MT SAFE modules. The framework accomplishes this by providing the MT STREAMS perimeter mechanisms for controlling and restricting the concurrency in a STREAMS module. See the section “MT SAFE Modules” on page 221.

The DDI/DKI entry points (`open`, `close`, `put`, and `service` procedures) plus certain callback procedures (scheduled with `qtimeout`, `qbufcall`, or `qwriter`) are synchronous entry points. All other entry points into a module are asynchronous.

Examples of the latter are hardware interrupt routines, `timeout`, `bufcall`, and `esballoc` callback routines.

STREAMS Framework Integrity

The STREAMS framework guarantees the integrity of the STREAMS data structures, such as `queue_t`, `mblk_t`, and `dblk_t`. This assumes that a module conforms to the DDI/DKI and does not directly access global operating system data structures or facilities not described within the Driver-Kernel Interface.

The `q_next` and `q_ptr` fields of the `queue_t` structure are not modified by the system while a thread is actively executing within a synchronous entry point. The `q_next` field of the `queue_t` structure could change while a thread is executing within an asynchronous entry point.

As in previous Solaris 2.x system releases, a module must not call another module's `put` or `service` procedures directly. The DDI/DKI routines **`putnext`**(9F), **`put`**(9F), and others 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.

When making your module MT SAFE, the integrity of private module data structures must be ensured by the module itself. Knowing the boundaries of what the framework supports is critical in deciding what you must provide yourself. The integrity of private module data structures can be maintained by either using the MT STREAMS perimeters to control the concurrency in the module, by using module private locks, or by a combination of the two.

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 **`putnext`**(9F), **`qreply`**(9F) returns.

MT Configurations

A module or a driver can be either MT SAFE or MT UNSAFE. Beginning with the 2.7 release of teh Solaris system, no MT UNSAFE module or driver will be supported.

MT SAFE modules

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.

It is easiest 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” on page 228provides a complete example of using a per-module perimeter, and “Sample Multithreaded Module with Outer Perimeter” on page 234 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. 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 226.

Independent of the perimeters, there will be at most one thread allowed within any given queue’s service procedure.

MT UNSAFE Modules

MT UNSAFE mode for STREAMS modules were temporarily supported as an aid in porting SVR4 modules. MT UNSAFE will not be supported in the next release of the operating system.

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

Note that much of the effort in conversion is simply determining the appropriate degree of data sharing and the corresponding granularity of locking. 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 accesses to those data structures. It is your responsibility to 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 module private data is single threaded with respect to the parts of the module that read the same data. Alternatively to the 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 that matches one of the available perimeters (or alternatively, requires the minimal number of locks) and that has a simple and straightforward implementation. Avoid additional complexity.

It is very common to overdo multithreading which results in a very low performance module.

Typical questions to answer 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?

Examples of natural levels of concurrency are:

- A module, where the `put` procedures read as well as modify module global data can be configured to be per-module single threaded using a per module inner perimeter.
- 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 for each queue (or queue pair) using the corresponding inner perimeter.
- 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, can be configured with a queue (or queue pair) inner perimeter plus an outer perimeter. The module can use `qwriter`to protect the sections where it modifies the module's global data.
- A module that falls in one of the previous categories, but requires higher concurrency for certain message types while not requiring message ordering, can be configured with the previous perimeters plus shared inner perimeter access for the `put` procedures. The module can then use `qwriter` when messages are handled in the `put` procedures that require exclusive access at the inner and/or outer perimeter.
- 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, these drivers restrict the concurrency when accessing the hardware registers in interrupt handlers. Such drivers need to be aware of the issues listed in "MT SAFE Modules using Explicit Locks" on page 226.

Porting to the SunOS 5.x System

When porting a STREAMS module or driver from the SunOS 4.x system to the SunOS 5.x system, the module should be examined with respect to the following areas:

- The SunOS5.x Device Driver Interface (DDI/DKI).
- The SunOS 5.x 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.x 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.x system should conform to the Device Driver Interface. The SunOS 5.x 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 details on the 5.x DDI and DVMA.

The kernel networking subsystem in the SunOS 5.X system is STREAMS based. Datalink drivers which used the `ifnet` interface in the SunOS 4.x system must be converted to use DLPI for the SunOS 5.X system. Refer to the Data Link Provider Interface, Revision 2 specification.

After reviewing the module for conformance to the SunOS 5.x DKI and DDI specifications, you should be able to consider the impact of multithreading on the module.

MT SAFE Modules

Your MT SAFE modules should use perimeters and avoid using module private locks. Should you opt to use module private locks you need to read “MT SAFE Modules using Explicit Locks” on page 226 in addition to this section.

MT STREAMS Perimeters

For the purpose of controlling and restricting the concurrency for the synchronous entry points, the STREAMS framework defines two MT *perimeters*. The STREAMS framework provides the concepts of *inner* and *outer* perimeters. A module can be configured either to have no perimeters, to have only an inner or an outer perimeter, or to both an inner and outer perimeter. For inner perimeters there are different scope perimeters to choose from. Unrestricted concurrency can be obtained by configuring no perimeters.

Figure Figure 12–1and Figure 12–2are examples of inner perimeters, and Figure 12–3shows multiple inner perimeters inside an outer perimeter.

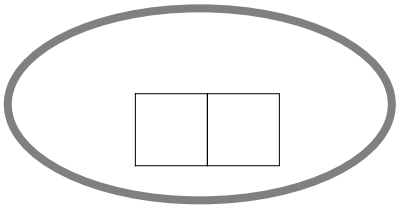


Figure 12–1 Inner Perimeter Spanning a Pair Of Queues. (`D_MP`TQAIR)

Both the inner and outer perimeters act as readers/writer locks allowing multiple readers or a single writer. Thus, each perimeter can be entered in two modes: shared (reader) or exclusive (writer). By default all synchronous entry points enter the inner perimeter exclusively and the outer perimeter shared.

The inner and outer perimeters are entered when one of the synchronous entry points is called and the perimeters are retained until the call returns from the entry point. Thus, for example, the thread does not leave the perimeter of one module when it calls `putnext` to enter another module.

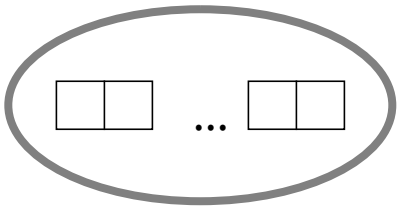


Figure 12–2 Inner Perimeter Spanning All queues In a Module. (`D_MTPERMOD`)

When a thread is inside a perimeter and it calls **`putnext`**(9F) (or **`putnextctl1`**(9F)), the thread can "loop around" through other STREAMS modules and try to reenter a put procedure inside the original perimeter. If this reentry conflicts with the earlier entry (for example if the first entry has exclusive access at the inner perimeter), the STREAMS framework defers the reentry while preserving the order of the messages attempting to enter the perimeter. Thus, **`putnext`**(9F) returns without the message having been passed to the `put` procedure and the framework passes the message to the `put` procedure when it is possible to enter the perimeters.

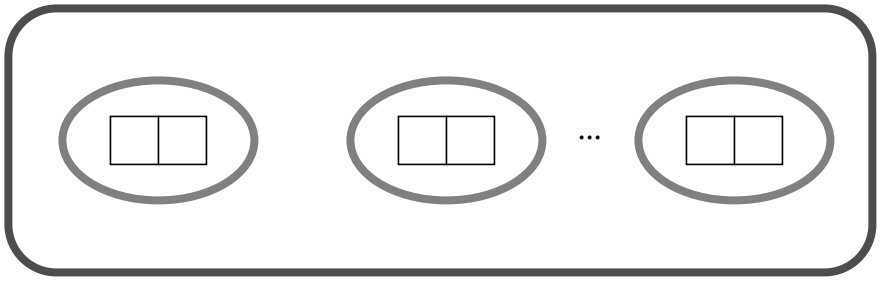


Figure 12–3 Outer Perimeter Spanning All Queues With Inner Perimeters Spanning Each Pair. (`D_MTOUTPERIM` Combined With `D_MTQPAIR`)

The optional outer perimeter spans all queues in a module (see Figure 12–3)

Perimeter options

Several flags are used to specify the perimeters. 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

The inner perimeter is controlled by these mutually exclusive flags:

- 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 of the above: The module has no inner perimeter.

The presence of the outer perimeter is configured using:

- D_MTOUTEPERIM: In addition to any 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.

Recall that by default all synchronous entry points enter the inner perimeter exclusively and enter the outer perimeter shared. This behavior can be modified in two ways:

- D_MTOCEXCL: The framework invokes the open and close procedures with exclusive access at the outer perimeter (instead of the default shared access at the outer perimeter.)
- D_MTPUTSHARED: The framework invokes the put procedures with shared access at the inner perimeter (instead of the default exclusive access at the inner perimeter.)

MT Configuration

To configure the driver as being MT SAFE, the **cb_ops**(9S)) and **dev_ops**(9S)) data structures must be initialized. This code must be in the header section of your module. For more information, see the example program in "Sample Multithreaded Device Driver" on page 228, **cb_ops**(9S)), and **dev_ops**(9S)).

The driver is configured to be MT SAFE by setting the cb_flag field to D_MP. It also configures any MT STREAMS perimeters by setting flags in the cb_flag field. (See **mt-streams**(9F). The corresponding configuration for a module is done using the f_flag field in **fmodsw**(9S)).

qprocson(9F)/qprocsoff(9F)

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.

qprocson(9F) must be called by the first `open` of a module, but only after allocation and initialization of any module resources on which the `put` and `service` procedures depend. The `qprocsoff`) routine must be called by the `close` routine of the module before deallocating any resources on which the `put` and `service` procedures depend.

To avoid deadlocks, modules must not hold private locks across the calls to **qprocson**(9F) or `qprocsoff(9F)`.

qtimeout(9F)/ **qunbufcall**(9F)

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 it complex for the callbacks to synchronize with the rest of the module.

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, hence with the same concurrency restrictions as the `put` and `service` procedures.

qwriter(9F))

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` or `service` procedures to upgrade to exclusive access at the outer perimeter. A module where the `put` procedures run with shared access at the inner perimeter (`D_MTPUTSHARED`) can use **qwriter**(9F) in the `put` or `service` procedures to upgrade to exclusive access at the inner perimeter.

Note - Note that **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 "Sample Multithreaded Module with Outer Perimeter" on page 234.

qwait(9F)

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) for an example.

Asynchronous Callbacks

Interrupt handlers and other asynchronous callback functions require special care by the module writer, since 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), since the `qtimeout` and `qbufcall` callbacks are synchronous and consequently introduce no special synchronization requirements.

Since 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 which the module's **timeout**(9F) or **bufcall**(9F) callback functions acquire. See section ""MT SAFE Modules using Explicit Locks" on page 226.
- Outstanding callbacks from **esballoc**(9F), if associated with a particular Stream, must be allowed to complete before the module `close` routine deallocates those private data structures on which they depend.

Close Race Conditions

Since 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.

Module Unloading and **esballoc**(9F)

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 has to refuse to be unloaded.

This is an issue mainly for modules and drivers using esballoc since 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 count of outstanding callbacks. They can refuse to be unloaded by having their **_fini**(9E) routine return `EBUSY` if there are outstanding callbacks.

Use of `q_next`

The `q_next` field in the `queue_t` structure can be dereferenced 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 reentering 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. Since `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 it is important to observe these constraints:

- Avoid holding module private locks across calls to **`putnext`**(9F), since the module might be reentered 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`) 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, since **`untimeout`**(9F)and **`unbufcall`**(9F) wait until an already executing callback has completed.

The first restriction makes it hard to use module private locks to presere message ordering. MT STREAMS perimeters is the preferred mechanism to preserve message ordering.

Preserving Message Ordering

Module private locks cannot be used to preserve message ordering, since 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, since the framework guarantees that at most one thread will execute in the `service` procedure for a given queue.

Use perimeters since there is a performance penalty to using `service` procedures.

Sample Multithreaded Device Driver

Code 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, at most 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 225.

CODE EXAMPLE 12–1 Sample Multithreaded, Loadable, STREAMS Pseudo-Driver

```
/*
 * Example SunOS 5.x 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 */
  int   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;

  /* do stuff here */
  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)) {
   /* do stuff 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 */
  /* do stuff */

}
```

Sample Multithreaded Module with Outer Perimeter

Code 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 since 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.

CODE EXAMPLE 12–2 Multithread Module with Outer Perimeter

```
/*
 * Example SunOS 5.x 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 */
  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 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;

  /*
   * Do stuff 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;

  /*
   * Do stuff 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;
  }
  /*
   * Do stuff 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)) {
  /*
   * Do stuff 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 */
  /*
   * Do stuff here. Can read ‘‘some_module_global_data’’ since we
   * have shared access at the outer perimeter.
   */
}
```

CHAPTER 13

Multiplexing

Overview of Multiplexing

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 user level by system calls.

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 the multiplexing facility. 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 was done for the loop-around driver and for the driver handling multiple minor devices in Chapter 9. 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(2) and close(2) 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, 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, it is not possible to provide general-purpose multiplexer drivers. Rather, STREAMS provides a general purpose multiplexing facility. The facility allows users to set up the intermodule or driver plumbing to create multiplexer configurations of generally unlimited interconnection.

Building a Multiplexer

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 will be 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 other 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.

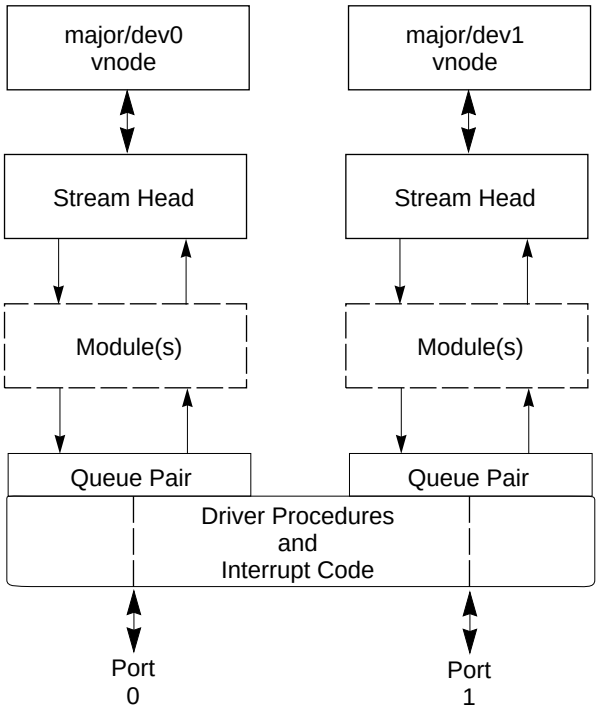


Figure 13–1 Protocol Multiplexer

Code Example 13–1 shows how this daemon process sets up the protocol multiplexer. The necessary declarations and initialization for the daemon program follow.

CODE 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. So, 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 Solaris 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 inFigure 13–2. The Stream to the 802.4 driver may now be connected below the multiplexing IP driver using the `I_LINK` **`ioctl`**(2).

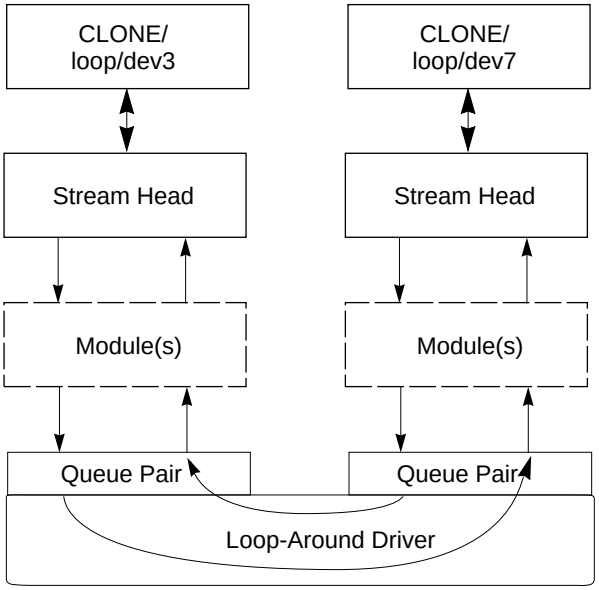


Figure 13–2 Before Link

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 is 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 later.

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 later in this chapter. 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 it is necessary to have intermediate modules in the Stream 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 (see *SunOS Reference Manual*, **`Intro`**(7)) describes such restrictions. 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 inFigure 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();
}
```

To summarize a multilevel protocol multiplexer. 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)s 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 the following section.

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. That is because **`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 example as a daemon process.

If the process uses persistent links through `I_PLINK` **`ioctl`**(2), the multiplexer configuration would remain intact after the process exits. “Persistent Links” on page 259 are described later in this chapter.

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 is driver dependent. For example, the protocol multiplexer might use address information found in a protocol header to determine over which subnetwork 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 to which Stream 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 could 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 could 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 devices, 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 conceptually divided into two parts: the lower half (bottom) and the upper half (top). The multiplexer queue structures that have been 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 bottom half **`qinit`**(9S) structures of 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 first two streamtab structure entries to create 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, ultimately, 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` message (acknowledgment). `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 `l_qbot` as the queue value. 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.

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 will automatically dismantle 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 a user(s) in the system. The net modules would 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. It is the control Stream, leading to a process that 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

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), flow control requires the use of standard and special code, since 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. Multiprocessor and multithreading issues such as locking for data corruption and to prevent race conditions are not discussed. See Chapter 12 for details.

Code Example 13–2 is of multiplexer declarations:

CODE 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 (in this case there is only one) lower Stream head 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.

Code Example 13–3, the upper queue open, contains the canonical driver open code:

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 the following: 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

`muxuwput`, the upper-queue write put procedure, traps `ioctl`s, in particular `I_LINK` and `I_UNLINK`:

```
static int
/*
* This is our callback routine usesd 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 bottom 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 **`ioctl`**(2)s are only accepted on this Stream. As described previously, a controlling Stream is the one that issues the `I_LINK`. Having a single control Stream is a recommended practice. `I_LINK` and `I_UNLINK` include a `linkblk` structure containing:

`l_qtop`

The upper write queue from which the **`ioctl`**(2) is coming. It always equals `q` for an `I_LINK`, and `NULL` for `I_PLINK`.

`l_qbot`

The new lower write queue. It is the former Stream-head write queue. It is of most interest since that is where the multiplexer gets and puts its data.

`l_index`

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 acknowledgment 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 allows the driver to send messages downstream to perform any initialization functions. Returning an `M_IOCNAK` message (negative acknowledgment) 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 acknowledgment. A negative acknowledgment should not be returned to an `I_UNLINK`. The Stream head assures 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)s (see “Persistent Links” on page 259) 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 would, 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

The following example shows the code for the 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:

CODE EXAMPLE 13–4

```
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. It handles `M_FLUSH` messages. Note 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 sanity checks are made:

- Check whether the device is in range
- Check whether the upper Stream is open
- Check whether the upper Stream is not full

This multiplexer does not support flow control on the read side. It is merely a router. If it passes all sanity checks, the message is put to the proper upper queue. Otherwise, the message is discarded.

The upper Stream close routine simply 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

With I_LINK and I_UNLINK **ioctl**(2)s 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 will dismantle the whole multiplexing configuration. It is not always desirable to keep a process running merely to hold the multiplexer configuration together. 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 will remain intact.

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 in the following way:

```
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 MUXdriver and send data to the Driver1 since the persistent link to the Driver1 remains intact.

The I_PUNLINK **ioctl**(2) is used dismantle the persistent link. Its format is:

```
ioctl(fd0, I_PUNLINK, muxid)
```

where the 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. The I_PUNLINK removes the persistent link between the multiplexer referenced by the 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 the 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 **ioctl**(2) with the I_LINK and I_UNLINK. Any attempt to unlink a regular link with the I_PUNLINK or to unlink a

persistent link with the I_UNLINK **ioctl**(2) causes the errno value of EINVAL to be returned.

Since multilevel multiplexing configurations are allowed in STREAMS, it is possible to have a situation where persistent links 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 creating a persistent link of multiplexer 2 below multiplexer 1 and then closing the controlling file descriptor associated with the multiplexer 2 and reopening it again and then 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, thus creating an infinite loop or overflowing 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 will 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.

PART III Advanced Topics

CHAPTER 14

STREAMS-Based Terminal Subsystem

This chapter describes how a terminal subsytem is set up, indicates how interrupts are handled. Different protocols are addressed, as well as canonical processing and line discipline substitution.

Overview of Terminal Subsystem

STREAMS provides a uniform interface for implementing character I/O devices and networking protocols in the kernel. SunOS 5.6 implements the terminal subsystem in STREAMS. The STREAMS-based terminal subsystem (Figure 14–1) 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 Sun 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 274).
- Merge with networking. By pushing a line discipline module on a network line, you can make the network look like a terminal line.

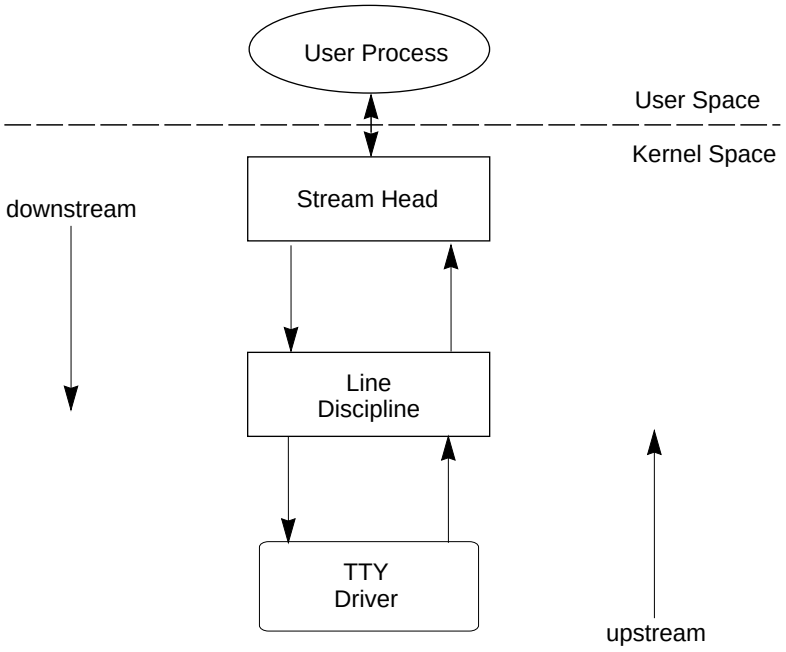


Figure 14–1 STREAMS-based Terminal Subsystem

The initial setup of the STREAMS-based terminal subsystem is handled with the **ttymon**(1M) command within the framework of the Service Access Facility or the autopush feature. See “ **autopush**(1M) Facility” on page 211 for more information on autopush.

The STREAMS-based terminal subsystem supports **termio**(7I), the **termios**(3) specification of the POSIX standard, multiple byte characters for internationalization, the interface to asynchronous hardware flow control (see **termio**(7I)), 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. However, the `sxt` feature is being discontinued and users are encouraged to use the job-control mechanism. Note that both `shl` and job control should *not* be run simultaneously.

Line-Discipline Module

A STREAMS line-discipline module called ldterm (see **ldterm**(7M)) is a key part of the STREAMS-based terminal subsystem. Throughout this chapter, the terms line discipline and **ldterm**(7M) 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 that was traditionally done through the `linesw` mechanism.

The **termio**(7I) and **termios**(3) 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 on and off the processing of these flags. 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 ldterm's default processing. By default, **ldterm**(7M) assumes that it must process all flags except `c_cflag`, unless it receives a message telling 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 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.

User-Configurable Settings

See **termio**(7I) for more information.

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 TTY's 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 and low water marks to 1024 and 200, respectively. These are the current values (although they may change over time).

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 (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 and low water marks for the read queue are 1024 and 200, respectively. These are the current values and may be subject to change.

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.

In canonical mode 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). It is possible to canonize the raw data, when the user has switched from raw to canonical mode. 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 down. 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. Since the read side high 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 by generating 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 **ioctl**(2)s 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_STOPI, M_STARTI, 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 message 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**(3) structure. The driver then turns on those c_iflag, c_oflag, and c_lflag fields of the **termios**(3) structure that are processed on the peripheral device by ORing 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.x is that AT&T's line discipline module does not check if 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.x.

EUC Handling in ldterm

The idea of letting post-processing (the o_flags) happen off the host processor is not recommended unless the board software is prepared to deal with international (EUC) character sets properly. The reason for this is that post-processing must take the EUC information into account. **ldterm**(7M) knows about the 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 one is using single byte 8-bit codes and has 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 a previous conditions, 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, it is checked to see if it is SS2 or SS3. If so, 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 for each logical character in the canonical buffer its display width. During erase processing, positions in the parallel array are consulted to determine how many backspaces need to 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 via a mechanism such as **getwidth**(3C) or **setlocale**(3C) and then copy the values into the structure in eucioctl.h, and send the structure via 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 allows **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 the environment in SunOS 5.x systems.

The file euc.h has the structure with fields for EUC width, screen width, and wide character width. The following functions are used to set and get EUC widths (these functions assume the environment where the eucwidth_t structure is needed and available):

CODE EXAMPLE 14–1 EUC

```
#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]);
}
```

For more detailed descriptions, see *System Interface Guide.*

Hardware Emulation Module

If a Stream supports a terminal interface, a driver or module that understands all ioctls to support terminal semantics (specified by **termio**(7I) and **termiox**(7I) is needed. 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 ioctls that may be sent to the process at the Stream head and to mediate the passage of control information downstream. The combination of the line- discipline module and the hardware emulation module behaves as if there were 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 (this is discussed later in this chapter), 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 ioctls 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)s.
- 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 any other **ioctl**(2)s are received on its write queue, it sends an `M_IOCNAK` message upstream. It doesn't pass any unrecognized **ioctl**(2)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 all devices, which 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 14–2 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 later).

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 a **`ioctl`**(2), such as `TCGETA` or `TCGETS`, is issued to indicate that canonical processing is being performed.

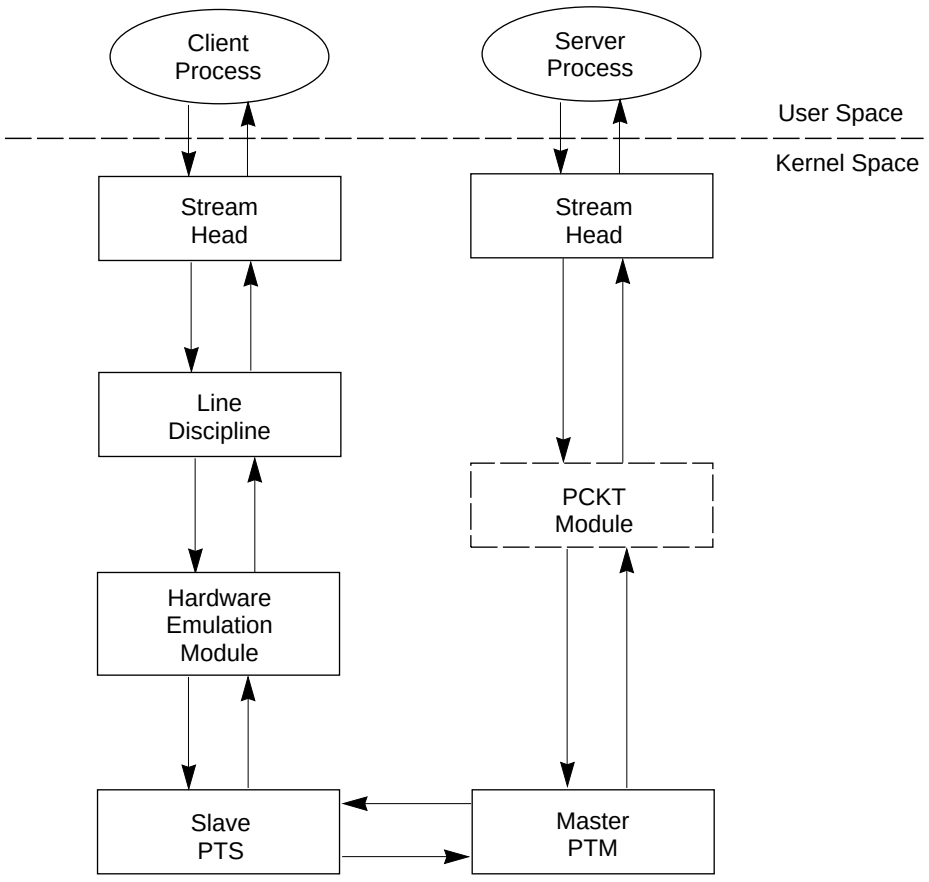


Figure 14–2 Pseudo-TTY Subsystem Architecture

Pseudo-TTY Emulation Module - **ptem**(7M)

Since 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)s 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**(2) 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. When setting the baud rate to 0, it has the effect of hanging up the pseudo-terminal.

The pseudo-terminal has no concept of 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 not also 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`**(2)s.
- When it receives any other **`ioctl`**(2) on its write queue, it sends an `M_IOCNAK` message upstream.
- It passes downstream the following **`ioctl`**(2)s after processing them: `TCSETA`, `TCSETAW`, `TCSETAF`, `TCSETS`, `TCSETSW`, `TCSETSF`, `TCSBRK`, and `TIOCSWINSZ`.
- **`ptem`**(7M) frees any `M_IOCNAK` messages it receives on its read queue in case the `pckt` module (**`pckt`**(7M) is described in the section "Packet Mode" on page 278) is not on the pseudo terminal subsystem and the above **`ioctl`**(2)s get to the master's Stream head which would then send an `M_IOCNAK` message.
- In its open routine, **`ptem`**(7M) sends an `M_SETOPTS` message upstream requesting allocation of a controlling TTY.
- When **`ptem`**(7M) 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`**(7M) 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 it 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`**(7M) 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 / disable canonical processing.
- When it receives an `M_DELAY` message on its read or write queue, it discards the message and does not act on it.
- When it receives an `M_IOCTL` of type `JWINSIZE` on its write queue and if the values in the `jwinsize` structure of **`ptem`**(7M) 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 it receives an `M_IOCTL` message of type `TIOCGWINSZ` on its write queue and if 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 if 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 not mentioned above are passed to the next module or driver.

Data Structure

SunOS 5.x reserves the right to change **ptem**(7M)'s internal implementation. This structure should be relevant only to people wanting to change the module.

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.

Open and Close Routines

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 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 can not be allocated (no buffer is available).
- A **stroptions**(9S) structure can not 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

A feature known as *remote mode* is available with the pseudo-TTY subsystem. This feature is used for applications that perform the canonical function normally done by **ldterm**(7M) and TTY driver. The remote mode allows 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 no canonical processing is to take place. The remote mode may be disabled by `ioctl(fd, TIOCREMOTE, 0)`.

Packet Mode

The STREAMS-based pseudo-terminal subsystem also supports a feature called *packet mode*. This 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.
- It 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), packetized 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**(7D) and **pts**(7D)

To use the pseudo-TTY subsystem, a node for the master side driver /dev/ptmx and N number of slave drivers (N is determined at installation time) must be installed. 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 that will 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 superuser 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` flag and vice versa.

When the master device is closed, an `M_HANGUP` message is sent to the slave device which will 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 will be 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. Since 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)s that are used by the functions **grantpt**(3C), **unlockpt**(3C), and **ptsname**(3C). The ISPTM **ioctl**(2).

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 input edited by the ldterm module 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.

To summarize, 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.

grantpt(3C)

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(3C)

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(3C)

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.

CHAPTER 15

Debugging

Overview of Debugging Facilities

This appendix describes some of the tools available to assist in debugging STREAMS-based applications.

The basic categories available for debugging can be broken into these following areas:

- Kernel debug printing - kernel facilities for printing from inside drivers
- STREAMS error logging - a STREAMS-supported model of generating error messages and allowing them to be received by one of three different types of loggers.
- Kernel-examination tools - which include the tools bundled with the operating system

Kernel Debug Printing

Console Messages

The kernel routine **cmn_err**(9F) allows printing of formatted strings on a system console. It displays a specified message on the console and/or stores it in the `putbuf` that is a circular array in the kernel and contains output from **cmn_err**(9F). Its format 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`**(3S). It is used to continue another message or to display an informative message not associated with an error.
- `CE_NOTE` - report 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. For example, a sector on a disk needing to be accessed repeatedly before it can be accessed correctly might be such an event.
- `CE_WARN` - 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` - 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, 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 `putbuf`. `putbuf` can be accessed with the **`crash`**(1M) 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 putbuf 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 putbuf 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.

STREAMS Error Logging

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/drivers and 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 is switched to the logger(s) 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

crash(1M) Command

crash(1M) examines kernel structures interactively. It can be used on a system dump and on an active system. 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), and 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**(1M) 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, `queue`s, and `stream`s. On the output report, the `CONFIG` column represents the number of structures currently configured. It may change because resources are allocated as needed.

adb(1) Command

adb(1) 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.

kadb(1M) Command

kadb(1M) is an interactive debugger with a user interface similar to **adb**(1), but runs in the same virtual address space as the program begin debugged. It also has no specific STREAMS support.

APPENDIX **A**

Message Types

Introduction

Defined STREAMS message types differ in their intended purposes, their treatment at the Stream head, and in 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.

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. 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.

Ordinary Messages

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

Contain 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 will 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 ticks 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 **ioctl**(2)s (see **streamio**(7I)). Also generated in response to ioctl calls that contain a command argument value not defined in **streamio**(7I). When one of these **ioctl**(2).

ioctl(2)s 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, and sends the message downstream. M_IOCTL messages are intended to 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 acknowledgment 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 passed in and, 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.

```
struct iocblk
{
 int ioc_cmd;        /* ioctl command type */
 cred_t *ioc_cr;        /* full credentials */
 uint ioc_id;        /* ioctl identifier */
 uint ioc_count;        /* cnt for ioctl data */
 int ioc_error;        /* M_IOCACK or M_IOCNAK */
 int ioc_rval;        /* ret val for M_IOCACK */
};
```

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.h>). Its contents can be tested to determine if 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 acknowledgment) or an M_IOCNAK (negative acknowledgment) 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 assuring 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.

If a module processing a transparent **ioctl**(2) wants to send data to a user process, it 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 free the blocks.

No data can be sent with an M_IOCNAK message for any type of M_IOCTL. The Stream head will ignore and will free 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

Used by STREAMS to pass 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

Intended to contain control information and associated data. The message format is one or more (see note) M_PROTO message blocks followed by zero or more M_DATA message blocks. The semantics of the M_DATA and M_PROTO message block are determined by the STREAMS module that receives the message.

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.

Note - On the write-side, the user can only generate M_PROTO messages containing one M_PROTO message block.

Although its use is not recommended, 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. **getmsg**(2)) compacts the blocks into a single control part when passing them to the user process.

M_RSE

Reserved for internal use. Modules that do not recognize this message must pass it on. Drivers that do not recognize it must free it.

M_SETOPTS

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 {
 ulong   so_flags;      /*options to set*/
 short   so_readopt;      /*read option*/
 ushort   so_wroff;      /*write offset*/
 long   so_minpsz;      /*min read packet size*/
 long   so_maxpsz;      /*max read packet size*/
 ulong   so_hiwat;      /*rd que hi-water mark*/
 ulong   so_lowat;      /*rd que low-water mark*/
>>>>>>>>>>>>>> unsigned char so_band;  /* upd water marks*/
};
```

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 is intended to limit the size of M_DATA messages that the module should put to the Stream head. There is no intended minimum size for other message types. The default value in the Stream head is zero.
- SO_MAXPSZ: 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 is intended to limit the size of M_DATA messages that the module should put to the Stream head. There is no intended maximum size for other message types. The default value in the Stream head is INFPSZ, the maximum 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 will have 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 has 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) will block on flow control and will block if buffers are not available.

  If both SO_NDELON and SO_NDELOFF are set in so_flags, SO_NDELOFF will have precedence.

Note - For conformance with the POSIX standard, it is recommended that new applications 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 and part of the buffer has been written and a flow control or buffers not available condition is encountered, **write**(2) terminate and return the number of bytes written.

  If O_NDELAY (or O_NONBLOCK) is clear, a **write**(2) will block on flow control and will block if buffers are not available.

  This is the default. If both SO_NDELON and SO_NDELOFF are set in so_flags, SO_NDELOFF will have precedence.

  In the STREAMS-based pipe mechanism, the behavior of **read**(2) and **write**(2) is different for the O_NDELAY and O_NONBLOCK flags.

- 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 the so_hiwat and so_lowat fields pertain to.

  If the SO_BAND flag is not set and the SO_HIWAT and SO_LOWAT flags are on, the normal high and low watermarks are affected. The SO_BAND flag has no effect if SO_HIWAT and SO_LOWAT flags are off.

  Only one band's watermarks 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 or may not be allocated as a controlling terminal via an M_SETOPTS message arriving upstream during open processing. If the Stream head is opened before receiving this message, the Stream will not be allocated as a controlling terminal until it is queued again by a session leader.

- SO_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

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 process(es) under the following conditions:

- If the signal is SIGPOLL, it will be 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 **ioctl**(2)s 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

M_COPYIN

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>.

```
struct copyreq {
  int   cq_cmd;    /* ioctl cmd (fr ioc_cmd) */
  cred_t  *cq_cr;    /* full credentials */
  uint  cq_id;    /* ioctl id (from ioc_id) */
  caddr_t cq_addr;       /* addr to copy data */
  uint  cq_size;    /* # bytes to copy */
  int  cq_flag;    /* reserved */
  mblk_t  *cq_private;    /* private state info */
};
```

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 will guarantee 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.

The cq_private field can be used by a module to point to a message block containing the module's state information relating to this **ioctl**(2). The Stream head copies (without processing) the contents of this field to the M_IOCDATA response message so that the module can resume the associated state. If an M_COPYIN or M_COPYOUT message is freed, STREAMS does not free any message block pointed to by cq_private. This is the module's responsibility.

This message should not be queued by a module or driver unless it intends to process the data for the **ioctl**(2).

M_COPYOUT

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.

M_ERROR

Sent upstream by modules or drivers to report some 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), fails with errno set to the first data byte of the message. POLLERR is set if the Stream is being **poll**(2)ed. 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.

The second style of the M_ERROR message is two bytes long. The first byte is the read error and the second byte is the write error. This allows 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. This allows a module to just 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

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 queue(s), 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

Sent upstream by a driver to report that it can no longer send data upstream. As 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 **ioctl**(2)s that cause messages to be sent downstream are also failed. POLLHUP is set if the Stream is being **poll**(2)ed.

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, since the allocation and deallocation of controlling terminals to a session is the responsibility of the controlling process.

M_IOCACK

Signals the positive acknowledgment 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 can not be returned in an M_IOCACK message responding to a transparent M_IOCTL. The data must have been sent with preceding M_COPYOUT message(s). If any M_DATA blocks follow the M_IOCACK block, the Stream head will ignore and free them.

The format and use of this message type is described further under M_IOCTL.

M_IOCDATA

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>.

```
struct copyresp {
 int   cp_cmd;       /* ioctl cmd (fr ioc_cmd) */
 cred_t *cp_cr;          /* full credentials */
 uint   cp_id;       /* ioctl id (from ioc_id) */
 caddr_t    cp_rval;       /* status of request */
 mblk_t   *cp_private;       /* state info */
};
```

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 will be 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

Signals the negative acknowledgment (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 will ignore and free 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 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 will allow 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 silently 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

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 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 one generates an M_SIG message when the DEL (delete) key is pressed on the terminal and one has already typed ahead, the M_SIG message becomes queued and the user doesn't get the call until it's too late; it becomes impossible to kill or interrupt a process by pressing a delete key.

M_READ

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

Request devices to start or stop their output. They are intended to produce momentary pauses in a device's output, not to turn devices on or off.

The message format 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

As SO_MREADOFF and M_STOP except that SO_MREADOFFI and M_STOPI are used to start and stop input.

M_UNHANGUP

Used to reconnect carrier after it has been dropped.

APPENDIX **B**

STREAMS Utilities

Kernel Utility Interface Summary

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**(9F)
canput(9F)	test for room in a queue
copyb(9F)	copy a message block

ROUTINE	DESCRIPTION
copymsg(9F)	copy a message
datamsg(9F)	test whether message is a data message
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
getq(9F)	get a message from a queue
insq(9F)	put a message at a specific place in a queue

ROUTINE	DESCRIPTION
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 mate 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

ROUTINE	DESCRIPTION
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**(9F) 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
strqget(9F)	obtain information on a queue or a band of the queue

ROUTINE	DESCRIPTION
strqset(9F)	change information on a queue or a band of the queue
testb(9F)	check for an available buffer
unbufcall(9F)	cancel **bufcall**(9F) 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

APPENDIX **C**

STREAMS F.A.Q.

This appendix provides answers to additional frequently asked questions.

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* STEAMS) an in-kernel filter is *much* faster.

Here are some Frequently asked IP interface questions, and answers

Q) Is there documentation that describes the interface between IP and network drivers, namely, the SUN specific requirements not outlined in the DLPI Version 2 specification?

A) IP is a STREAMS module in Solaris 2.X. 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.

Q) 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 Style 2 provider?

A) Please look at '`dl_info_ack_t`' struct in `/usr/include/sys/dlpi.h`.

Q) Is it possible for the driver to be a `CLONE` driver and also a DLPI Style 2 provider?

A)Yes.

Q) 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 not valid prior to the `DL_ATTACH_REQ`. How do you suggest this be handled?

A) 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. I don't think the DL_ATTACH_REQ request has 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. To the question of mapping minor number to instance, usually the minor number (**getminor**(9F)) is the instance number.

Q) 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?

A) For the CLONE driver, I don't know if it is possible to do that. For the non-CLONE driver, it is possible to use the bits information in a particular minor number, for example FF, to map all other minor nodes.

Q) Does Solaris 2.1 ethernet drivers support LLI 2.0 interfaces?

A) Do you mean 'DLPI' (Data Link Provider interfaces) ? The Solaris 2.1 ethernet drivers, le and ie. both support DLPI. Please see man pages of le and ie.

Q) Does Solaris 2.1 DLPI provide both connection oriented services and connection less services. Also, is your DLPI Version 2.0, which includes multicast facilities.

A) Yes and yes. Please see man page of 'dlpi'.

Q) Is multicasting supported on SunOS 4.x? If not, how can the customer obtain this feature?

A) IP multicast is a standard supported feature in Solaris 2.x, but we don't support it in SunOS 4.x. If the customer wants to run an unsupported IP multicast on his/her SunOS 4.x machines, it can be got from public domain object distribution that Steve Deering, the inventor of IP multicast, distributes. This is available via anonymous FTP from gregorio.stanford.edu in the file vmtp-ip/ipmulti-sunos41x.tar.Z.

Glossary

autopush A STREAMS mechanism that enables a pre-specified 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 watermark.

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(2) 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 on SPARC® hardware.

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 towards a driver. Also called write-side and output side.

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.

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 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 a Stream. This module understands all ioctls necessary to support terminal semantics specified by termio(7) and termios(7).

input side A direction of data flow going from a driver towards the Stream head. Also called read-side and upstream.

line discipline A STREAMS module that performs termio(7) canonical and non-canonical processing. It shares some termio(7) 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 allows 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 via a call to fattach(3C) (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(2) 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 Same as a 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 a 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 which 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-side A direction of data flow going from a driver towards the Stream head. Also called upstream and input side.

read queue A message queue in a module or driver containing messages moving upstream. Associated with the read(2) system call and input from a driver.

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 which will subsequently have their service procedure called by the STREAMS scheduler. STREAMS scheduling is independent of the Solaris 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 in 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 a unidirectional flow of data between two processes where data written by one process become 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.

STREAMS-based pipe A mechanism used for bidirectional data transfer implemented using STREAMS, and sharing the properties of STREAMS-based devices.

Stream end A Stream component furthest from the user process, containing 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 comprises integral functions, utility routines, kernel facilities, and a set of structures.

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.

watermark A limit value used in flow control. Each queue has a high watermark and a low watermark. The high watermark value indicates the upper limit related to the number of bytes contained on the queue. When the queued character reaches its high watermark, 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 watermark value, the other queue will be unblocked by STREAMS.

write queue A message queue in a module or driver containing messages moving downstream. Associated with the write(2) system call and output from a user process.

write-side A direction of data flow going from the Stream head towards a driver. Also called downstream and output side.

Index

N

O

T

U

W