AT&T

UNIX® System V/386

Release 3.2

STREAMS PROGRAMMER'S GUIDE

UNIX
System V/386

UNIX® System V/386 Release 3.2

STREAMS PROGRAMMER'S GUIDE

PRENTICE
HALL

AT&T

UNIX® System V/386 Release 3.2

Streams Programmer's Guide

Prentice Hall, Englewood Cliffs, New Jersey 07632

Library of Congress Catalog Card Number: 88-62530

Editorial/production supervision: Karen Skrable Fortgang
Manufacturing buyer: Mary Ann Gloriande

© 1989 by AT&T. All rights reserved.
Published by Prentice-Hall, Inc.
A Division of Simon & Schuster
Englewood Cliffs, New Jersey 07632

All rights reserved. No part of this book may be
reproduced, in any form or by any means,
without permission in writing from the publisher.

NOTICE
The information in this document is subject to change without notice.
AT&T assumes no responsibility for any errors that may appear in
this document.

ETHERNET is a trademark of Xerox Corporation.
Intel is a registered trademark of Intel Corporation.
UNIX is a registered trademark of AT&T.
Xerox is a registered trademark of Xerox Corporation.

The publisher offers discounts on this book when ordered
in bulk quantities. For more information, write or call:

Special Sales
Prentice-Hall, Inc.
College Technical and Reference Division
Englewood Cliffs, NJ 07632
(201) 592-2498

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-944919-1

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

Table of Contents

Application Programming

Introduction to Part 1 xi

1 Basic Operations

A Simple Stream 1-1
Inserting Modules 1-4
Module and Driver Control 1-7

2 Advanced Operations

Advanced Input/Output Facilities 2-1
Input/Output Polling 2-2
Asynchronous Input/Output 2-7
Clone Open 2-8

3 Multiplexed Streams

Multiplexer Configurations 3-1
Building a Multiplexer 3-4
Dismantling a Multiplexer 3-13
Routing Data Through a Multiplexer 3-14

4 Message Handling

Service Interface Messages 4-1

The Message Interface 4-4
Datagram Service Interface Example 4-7

Module and Driver Programming

Introduction to Part 2 i

5 Streams Mechanism

Overview 5-1
Stream Construction 5-3
Opening a Stream 5-5
Adding and Removing Modules 5-6
Closing 5-7

6 Modules

Module Declarations 6-1
Module Procedures 6-4
Module and Driver Environment 6-6

7 Messages

Message Format 7-1
Filter Module Declarations 7-5
Message Allocation 7-8
Put Procedure 7-9

8 Message Queues and Service Procedures

The queue_t Structure 8-1
Service Procedures 8-2

Message Queues and Message Priority 8-4
Flow Control 8-6
Example 8-8

9 **Drivers**
Overview of Drivers 9-1
Driver Flow Control 9-3
Driver Programming 9-4
Driver Processing Procedures 9-8
Driver and Module Ioctls 9-12
Driver Close 9-15

10 **Complete Driver**
Cloning 10-1
Loop-Around Driver 10-2

11 **Multiplexing**
Multiplexing Configurations 11-1
Multiplexer Construction Example 11-5
Multiplexing Driver 11-8

12 **Service Interface**
Definition 12-1
Example 12-3

13 **Advanced Topics**
Recovering From No Buffers 13-1
Advanced Flow Control 13-6

Signals 13-7
Control of Stream Head Processing 13-9

A Appendix A: Kernel Structures

Appendix A: Kernel Structures A-1

B Appendix B: Message Types

Appendix B: Message Types B-1
Ordinary Messages B-2
Priority Messages B-10

C Appendix C: Utilities

Appendix C: Utilities C-1
Utility Descriptions C-3
Buffer Allocation Priority C-16
Utility Routine Summary C-17

D Appendix D: Design Guidelines

Appendix D: Design Guidelines D-1

E Appendix E: Configuring

Appendix E: Configuring E-1

Glossary

Glossary G-1

Index

Index I-1

List of Figures

Figure 1-1: Stream to Communications Driver 1-3
Figure 1-2: Case Converter Module 1-6
Figure 3-1: Many-to-One Multiplexer 3-1
Figure 3-2: One-to-Many Multiplexer 3-2
Figure 3-3: Many-to-Many Multiplexer 3-2
Figure 3-4: Protocol Multiplexer 3-4
Figure 3-5: Before Link 3-6
Figure 3-6: IP Multiplexer After First Link 3-7
Figure 3-7: IP Multiplexer 3-8
Figure 3-8: TP Multiplexer 3-10
Figure 4-1: Protocol Substitution 4-2
Figure 4-2: Service Interface 4-3
Figure 5-1: Downstream Stream Construction 5-3
Figure 5-2: QUEUE Data Structures 5-4
Figure 7-1: Message Form and Linkage 7-2
Figure 8-1: Message Queue Priority 8-4
Figure 9-1: Device Driver Streams 9-2

List of Figures

Figure 10-1: Loop-Around Streams 10-3

Figure 11-1: Internet Multiplexer Before Connecting 11-5

Figure 11-2: Internet Multiplexer After Connecting 11-6

Figure 11-3: Example Multiplexer Configuration 11-12

Figure B-1: M_PROTO and M_PCPROTO Message Structure B-3

Preface

Introduction to this Guide xi
Notational Conventions xi

STREAMS Overview xii

Development Facilities xv

Introduction to this Guide

This document provides information to developers on the use of the STREAMS mechanism at user and kernel levels.

STREAMS was first incorporated in UNIX System V Release 3.1 to augment the existing character input/output (I/O) mechanism and to support development of communication services. The *STREAMS Programmer's Guide* includes detailed information, with various examples, on the development methods and design philosophy of all aspects of STREAMS.

This guide is organized into two parts. Part 1, Applications Programming, describes the development of user level applications. Part 2, Module and Driver Programming, describes the STREAMS kernel facilities for development of modules and drivers. Although chapter numbers are consecutive, the two parts are independent. Working knowledge of the *STREAMS Primer* is assumed.

Notational Conventions

The following notational conventions are used throughout this *Guide*:

bold	User input, such as commands, options to commands, and the names of directories and files, appear in **bold**.
italic	Names of variables to which values must be assigned (such as *filename*) appear in *italic*.
command(number)	A command name followed by a number in parentheses refers to the part of a UNIX System reference manual that documents that command. (There are two reference manuals: the *User's/System Administrator's Reference Manual* and the *Programmer's Reference Manual*.) For example, the notation *cat*(1) refers to the page in section 1 (of the *User's/System Administrator's Reference Manual*) that documents the **cat** command.
`constant width`	UNIX System output, such as prompt signs and responses to commands, and program examples appear in `constant width`.

STREAMS Overview

This section reviews the STREAMS mechanism. STREAMS is a general, flexible facility and a set of tools for development of UNIX System communication services. It supports the implementation of services ranging from complete networking protocol suites to individual device drivers. 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 associated mechanism is simple and open-ended. It consists of a set of system calls, kernel resources, and kernel routines.

The standard interface and mechanism enable modular, portable development and easy integration of higher performance network services and their components. STREAMS provides a framework; it does not impose any specific network architecture. The STREAMS user interface is upwardly compatible with the character I/O user interface, and both user interfaces are available in UNIX System V Release 3.1 and subsequent releases.

A Stream is a full-duplex processing and data transfer path between a STREAMS driver in kernel space and a process in user space (see Figure 1). In the kernel, a Stream is constructed by linking a stream head, a driver and zero or more modules between the stream head and driver. The Stream head is the end of the Stream closest to the user process. Throughout this guide, the word "STREAMS" will refer to the mechanism and the word "Stream" will refer to the path between a user and a driver.

A STREAMS driver may be a device driver that provides the services of an external I/O device, or a software driver, commonly referred to as a pseudo-device driver, that performs functions internal to a Stream. The Stream head provides the interface between the Stream and user processes. Its principal function is to process STREAMS-related user system calls.

Data is passed between a driver and the Stream head in messages. Messages that are passed from the Stream head toward the driver are said to travel downstream. Similarly, messages passed in the other direction travel upstream. The Stream head transfers data between the data space of a user process and STREAMS kernel data space. Data to be sent to a driver from a user process are packaged into STREAMS messages and passed downstream. When a message containing data arrives at the Stream head from downstream, the message is processed by the Stream head, which copies the data into user buffers.

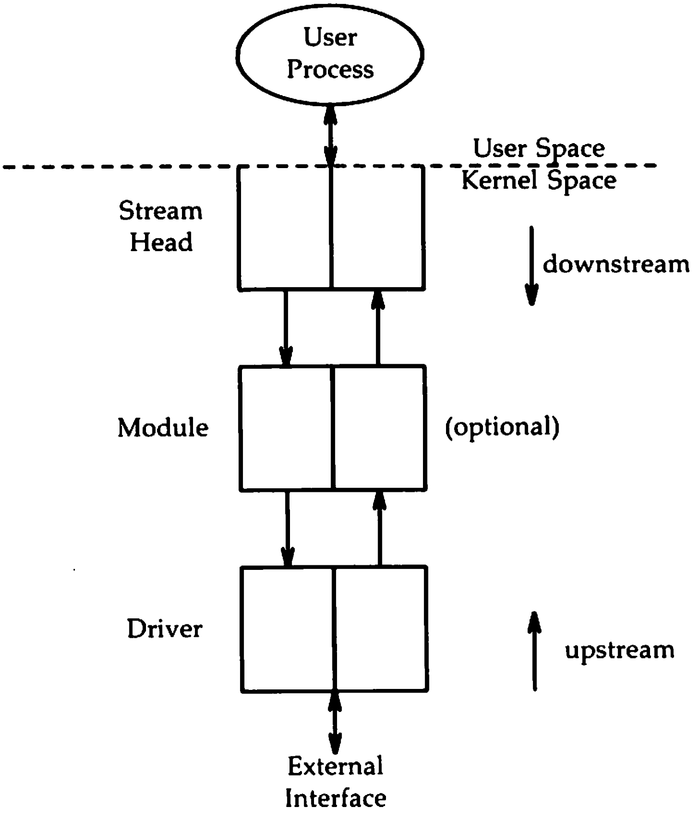
User Process
User Space
Kernel Space
Stream Head
downstream
Module
(optional)
Driver
upstream
External Interface

Figure 1: Basic Stream

Within a Stream, messages are distinguished by a type indicator. Certain message types sent upstream may cause the Stream head to perform specific actions, such as sending a signal to a user process. Other message types are intended to carry information within a Stream and are not directly seen by a user process.

One or more kernel-resident modules may be inserted into a Stream between the Stream head and driver to perform intermediate processing of data as it passes between the Stream head and driver. STREAMS modules are dynamically interconnected in a Stream by a user process. No kernel programming, assembly, or link editing is required to create the interconnection.

Development Facilities

General and STREAMS-specific system calls provide the user-level facilities required to implement application programs. This system call interface is upwardly compatible with the character I/O facilities. The **open** system call will recognize a STREAMS file and create a Stream to the specified driver. A user process can receive and send data on STREAMS files using **read** and **write** in the same manner as with character files. The **ioctl** system call enables users to perform functions specific to a particular device and a set of generic STREAMS **ioctl** commands [see *streamio*(7)] support a variety of functions for accessing and controlling Streams. A **close** will dismantle a Stream.

In addition to the generic **ioctl** commands, there are STREAMS-specific system calls to support unique STREAMS facilities. The **poll** system call enables a user to poll multiple Streams for various events. The **putmsg** and **getmsg** system calls enable users to send and receive STREAMS messages, and are suitable for interacting with STREAMS modules and drivers through a service interface.

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

Examples are used throughout both parts of this document to highlight the most important and common capabilities of STREAMS. The descriptions are not meant to be exhaustive. For simplicity, the examples reference fictional drivers and modules.

Appendix C provides the reference for STREAMS kernel utilities. STREAMS system calls are specified in Section 2 of the *Programmer's Reference Manual*. STREAMS utilities are specified in Section 1M of the *User's/System Administrator's Reference Manual*. STREAMS-specific **ioctl** calls are specified in *streamio*(7) of the *User's/System Administrator's Reference Manual*. The modules and drivers available are described in Section 7 of the *User's/System Administrator's Reference Manual*.

Introduction to Part 1

Part 1 of the guide, Application Programming, provides detailed information, with various examples, on the user interface to STREAMS facilities. It is intended for application programmers writing to the STREAMS system call interface. Working knowledge of UNIX System user programming, data communication facilities, and the *STREAMS Primer* is assumed. The organization of Part 1 is as follows:

- Chapter 1, Basic Operations, describes the basic operations available for constructing, using, and dismantling Streams. These operations are performed using **open**, **close**, **read**, **write**, and **ioctl**.
- Chapter 2, Advanced Operations, presents advanced facilities provided by STREAMS, including: **poll**, a user level I/O polling facility; asynchronous I/O processing support; and a new facility for sampling drivers for available resources.
- Chapter 3, Multiplexed Streams, describes the construction of sophisticated, multiplexed Stream configurations.
- Chapter 4, Message Handling, describes how users can process STREAMS messages using **putmsg** and **getmsg** in the context of a service interface example.

1 Basic Operations

A Simple Stream 1-1

Inserting Modules 1-4

Module and Driver Control 1-7

A Simple Stream

This chapter describes the basic set of operations for manipulating STREAMS entities.

A STREAMS driver is similar to a character I/O driver in that it has one or more nodes associated with it in the file system, and it is accessed using the **open** system call. Typically, each file system node corresponds to a separate minor device for that driver. Opening different minor devices of a driver will cause separate Streams to be connected between a user process and the driver. The file descriptor returned by the **open** call is used for further access to the Stream. If the same minor device is opened more than once, only one Stream will be created; the first **open** call will create the Stream, and subsequent **open** calls will return a file descriptor that references that Stream. Each process that opens the same minor device will share the same Stream to the device driver.

Once a device is opened, a user process can send data to the device using the **write** system call and receive data from the device using the **read** system call. Access to STREAMS drivers using **read** and **write** is compatible with the character I/O mechanism.

The **close** system call will close a device and dismantle the associated Stream.

The following example shows how a simple Stream is used. In the example, the user program interacts with a generic 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 can be retrieved by reading it from the device.

```
#include <fcntl.h>

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

   if ((fd = open("/dev/comm01", O_RDWR)) < 0) {
       perror("open failed");
       exit(1);
   }

   while ((count = read(fd, buf, 1024)) > 0) {
       if (write(fd, buf, count) != count) {
           perror("write failed");
           break;
       }
   }
   exit(0);
}
```

In the example, **/dev/comm01** identifies a minor device of the 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 1-1 shows the state of the Stream following the call to **open**.

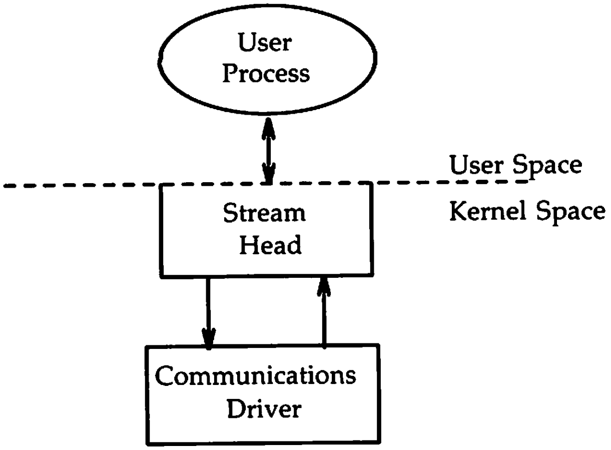

Figure 1-1: Stream to Communications Driver

This example illustrates a user reading data from the communications device and then writing the input back out to the same device. In short, this program echoes all input back over the communications line. The example assumes that a user is sending data from the other side of the communications line. The program reads up to 1024 bytes at a time, and then writes the number of bytes just read.

The **read** call returns the available data, which may contain fewer than 1024 bytes. If no data is currently available at the Stream head, the **read** call blocks until data arrives.

Similarly, the **write** call attempts to send *count* bytes to **/dev/comm01**. However, STREAMS implements a flow control mechanism that prevents a user from flooding a device driver with data, thereby exhausting system resources. If the Stream exerts flow control on the user, the **write** call blocks until the flow control has been relaxed. The call will not return until it has sent *count* bytes to the device. **exit** [see *exit*(2)] is called to terminate the user process. This system call also closes all open files, thereby dismantling the Stream in this example.

Inserting Modules

An advantage of STREAMS over the existing character I/O mechanism stems from the ability to insert various modules into a Stream to process and manipulate data that passes between a user process and the driver. The following example extends the previous communications device echoing example by inserting a module in the Stream to change the case of certain alphabetic characters. The case converter module is passed an input string and an output string by the user. Any incoming data (from the driver) is inspected for instances of characters in the module's input string and the alphabetic case of all matching characters is changed. Similar actions are taken for outgoing data using the output string. The necessary declarations for this program are shown below:

```
#include <string.h>
#include <fcntl.h>
#include <stropts.h>

/*
 * These defines would typically be
 * found in a header file for the module
 */
#define OUTPUT_STRING  1
#define INPUT_STRING   2

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

The first step is to establish a Stream to the communications driver and insert the case converter module. The following sequence of system calls accomplishes this:

```
if ((fd = open("/dev/comm01", O_RDWR)) < 0) {
    perror("open failed");
    exit(1);
}

if (ioctl(fd, I_PUSH, "case_converter") < 0) {
    perror("ioctl I_PUSH failed");
    exit(2);
}
```

The I_PUSH **ioctl** call directs the Stream head to insert the case converter module between the driver and the Stream head, creating the Stream shown in Figure 1-2. As with any driver, this module resides in the kernel and must have been configured into the system before it was booted. I_PUSH is one of several generic STREAMS **ioctl** commands that enable a user to access and control individual Streams [see *streamio*(7)].

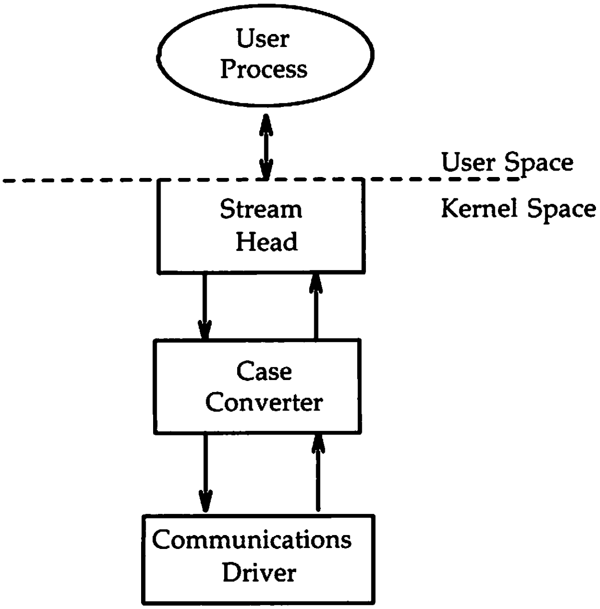

Figure 1-2: Case Converter Module

An important difference between STREAMS drivers and modules is illustrated here. Drivers are accessed through a node or nodes in the file system and may be opened just like any other device. Modules, on the other hand, do not occupy a file system node. Instead, they are identified through a separate naming convention and are inserted into a Stream using I_PUSH. The name of a module is defined by the module developer and is typically included on the manual page describing the module. (Manual pages describing STREAMS drivers and modules are found in section 7 of the *User's/System Administrator's Reference Manual.*)

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

Module and Driver Control

The next step in this example is to pass the input string and output string to the case converter module. This can be accomplished by issuing **ioctl** calls to the case converter module as follows:

```
/* set input conversion string */
strioctl.ic_cmd = INPUT_STRING; /* command type */
strioctl.ic_timout = 0;         /* default timeout (15 sec) */
strioctl.ic_dp = "ABCDEFGHIJ";
strioctl.ic_len = strlen(strioctl.ic_dp);

if (ioctl(fd, I_STR, &strioctl) < 0) {
    perror("ioctl I_STR failed");
    exit(3);
}

/* set output conversion string */
strioctl.ic_cmd = OUTPUT_STRING;/* command type */
strioctl.ic_dp = "abcdefghij";
strioctl.ic_len = strlen(strioctl.ic_dp);

if (ioctl(fd, I_STR, &strioctl) < 0) {
    perror("ioctl I_STR failed");
    exit(4);
}
```

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

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

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

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. The request will be processed by the module or driver closest to the Stream head that understands the command specified by *ic_cmd*. The **ioctl** call will block up to *ic_timout* seconds, waiting for the target module or driver to respond with either a positive or negative acknowledgment message. If an acknowledgment is not received in *ic_timout* seconds, the **ioctl** call will fail.

I_STR is actually a nested request; the Stream head intercepts I_STR and then sends the driver or module request (as specified in the **strioctl** structure) downstream. Any module that does not understand the command in *ic_cmd* will pass the message further downstream. Eventually, the request will reach the target module or driver, where it is processed and acknowledged. If no module or driver understands the command, a negative acknowledgment will be generated, and the **ioctl** call will fail.

In the example, two separate commands are sent to the case converter module. The first contains the conversion string for input data, and the second contains the conversion string for output data. The *ic_cmd* field is set to indicate whether the command is setting the input or output conversion string. For each command, the value of *ic_timout* is set to zero, which specifies the system default timeout value of 15 seconds. Also, a data argument that contains the conversion string accompanies each command. The *ic_dp* field points to the beginning of each string, and *ic_len* is set to the length of the string.

NOTE

Only one I_STR request can be active on a STREAM at one time. Further requests will block until the active I_STR request is acknowledged and the system call completes.

The **strioctl** structure is also used to retrieve the results, if any, of an I_STR request. If data is returned by the target module or driver, *ic_dp* must point to a buffer large enough to hold that data, and *ic_len* will be set on return to indicate the amount of data returned.

The remainder of this example is identical to the previous example:

```
    while ((count = read(fd, buf, 1024)) > 0) {
        if (write(fd, buf, count) != count) {
            perror("write failed");
            break;
        }
    }
    exit(0);
}
```

The case converter module will convert the specified input characters to lower case and the corresponding output characters to upper case. Notice that the case conversion processing was realized with *no* change to the communications driver.

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

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

2 Advanced Operations

Advanced Input/Output Facilities 2-1

Input/Output Polling 2-2

Asynchronous Input/Output 2-7

Clone Open 2-8

Advanced Input/Output Facilities

The traditional input/output facilities—**open**, **close**, **read**, **write**, and **ioctl**—have been discussed, but STREAMS supports new user capabilities that will be described in the remaining chapters of this guide. This chapter describes a facility that enables a user process to poll multiple Streams simultaneously for various events. Also discussed is a signaling feature that supports asynchronous I/O processing. Finally, this chapter presents a new mechanism, called **clone open**, for finding available minor devices.

Input/Output Polling

The **poll** [see *poll*(2)] system call provides users with a mechanism for monitoring input and output on a set of file descriptors that reference open Streams. It identifies those 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. These events include the following:

POLLIN	Input data is available on the Stream associated with the given file descriptor.
POLLPRI	A priority message is available on the Stream associated with the given file descriptor. Priority messages are described in the section of Chapter 4 entitled "Accessing the Datagram Provider."
POLLOUT	The Stream associated with the given file is writable. That is, the Stream has relieved the flow control that would prevent a user from sending data over that Stream.

poll will examine each file descriptor for the requested events and, on return, will indicate which events have occurred for each file descriptor. If no event has occurred on any polled file descriptor, **poll** blocks until a requested event or timeout occurs. The specific arguments to **poll** are the following:

- 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)

The following example shows the use of **poll**. Two separate minor devices of the communications driver presented earlier are opened, thereby establishing two separate Streams to the driver. Each Stream is polled for incoming data. If data arrives on either Stream, it is read and then written back to the other Stream. This program extends the previous echoing example by sending echoed data over a separate communications line (minor device). The steps needed to establish each Stream are as follows:

```
#include <fcntl.h>
#include <poll.h>

#define NPOLL 2         /* number of file descriptors to poll */

main( )
{
   struct pollfd pollfds[NPOLL];
   char buf[1024];
   int count, i;

   if ((pollfds[0].fd = open("/dev/comm01", O_RDWR|O_NDELAY)) < 0) {
       perror("open failed for /dev/comm01");
       exit(1);
   }

   if ((pollfds[1].fd = open("/dev/comm02", O_RDWR|O_NDELAY)) < 0) {
       perror("open failed for /dev/comm02");
       exit(2);
   }
```

The variable *pollfds* is declared as an array of **pollfd** structures, where this structure is defined in **<poll.h>** and has the following 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 will indicate which of the requested events has occurred.

The example opens two separate minor devices of the communications driver and initializes the *pollfds* entry for each. The remainder of the example uses **poll** to process incoming data as follows:

```
    /* set events to poll for incoming data */
    pollfds[0].events = POLLIN;
    pollfds[1].events = POLLIN;

    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 */
                perror("error event");
                exit(4);

            case 0:                                /* no events */
                break;

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

The user specifies the polled events by setting the *events* field of the **pollfd** structure to **POLLIN**. This requested event directs **poll** to notify the user of any incoming data on each Stream. The bulk of the example is an infinite loop, where each iteration will poll both Streams for incoming data.

The second argument to **poll** specifies the number of entries in the *pollfds* array (two in this example). The third argument is a timeout value indicating the number of milliseconds **poll** should wait for an event if none has occurred. On a system where millisecond accuracy is not available, *timeout* is rounded up to the nearest legal value available on that system. Here, the value of *timeout* is -1, specifying that **poll** should block indefinitely until a requested event occurs or until the call is interrupted.

If **poll** succeeds, the program looks at each entry in *pollfds*. If *revents* is set to 0, no event has occurred on that file descriptor. If *revents* is set to **POLLIN**, incoming data is available. In this case, 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 the only requested event was **POLLIN**. The following error events are defined for **poll**. These events may not be polled for by the user, but will be reported in *revents* whenever they occur. As such, they are only valid in the *revents* bitmask:

POLLERR	A fatal error has occurred in some module or driver on the Stream associated with the specified file descriptor. Further system calls will fail.
POLLHUP	A hangup condition exists on the Stream associated with the specified file descriptor.
POLLNVAL	The specified file descriptor is not associated with an open Stream.

The example attempts to process incoming data as quickly as possible. However, when writing data to a Stream, the **write** call may 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 flag set. If flow control is exerted and O_NDELAY is set, **write** will not be able to send all the data. This can occur if the communications driver is unable to keep up with the user's rate of data transmission. If the Stream becomes full, the number of bytes **write** sends will be less than the requested *count*. For simplicity, the example ignores the data if the Stream becomes full, and a warning is printed to **stderr**.

This program will continue until an error occurs on a Stream, or until the process is interrupted.

Asynchronous Input/Output

The **poll** system call described above enables a user to monitor multiple Streams in a synchronous fashion. The **poll** call 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 may wish to do some local processing and be interrupted when a pending event occurs. Some time-critical applications cannot afford to block, but must have immediate indication of success or failure.

A new facility is available for use with STREAMS that enables a user process to request a signal when a given event occurs on a Stream. When used with **poll**, this facility enables applications to asynchronously monitor a set of file descriptors for events.

The I_SETSIG **ioctl** call [see *streamio*(7)] is used to request that a SIGPOLL signal be sent to a user process when a specific event occurs. Listed below are the events for which an application may be signaled:

S_INPUT	Data has arrived at the Stream head, and no data existed at the Stream head when it arrived.
S_HIPRI	A priority STREAMS message has arrived at the Stream head.
S_OUTPUT	The Stream is no longer full and can accept output. That is, the Stream has relieved the flow control that would prevent a user from sending data over that Stream.
S_MSG	A special STREAMS signal message that contains a SIGPOLL signal has reached the front of the Stream head input queue. This message may be sent by modules or drivers to generate immediate notification of data or events to follow.

The polling example could be written to process input from each communications driver minor device by issuing I_SETSIG to request a signal for the **S_INPUT** event on each Stream. The signal catching routine could then call **poll** to determine on which Stream the event occurred. The default action for SIGPOLL is to terminate the process. Therefore, the user process must catch the signal using **signal** [see *signal*(2)]. SIGPOLL will only be sent to processes that request the signal using I_SETSIG.

Clone Open

In the earlier examples, each user process connected a Stream to a driver by opening a particular minor device of that driver. Often, however, a user process wants to connect a new Stream to a driver regardless of which minor device is used to access the driver.

In the past, this typically forced the user process to poll the various minor device nodes of the driver for an available minor device. To alleviate this task, a facility called **clone open** is supported for STREAMS drivers. If a STREAMS driver is implemented as a cloneable device, a single node in the file system may be opened to access any unused minor device. This special node guarantees that the user will be allocated a separate Stream to the driver on every **open** call. Each Stream will be associated with an unused minor device, so the total number of Streams that may be connected to a cloneable driver is limited by the number of minor devices configured for that driver.

The clone device may be useful, for example, in a networking environment where a protocol pseudo-device driver requires each user to open a separate Stream over which it will establish communication. Typically, the users would not care which minor device they used to establish a Stream to the driver. Instead, the clone device can find an available minor device for each user and establish a unique Stream to the driver. Chapter 3 describes this type of transport protocol driver.

NOTE

A user program has no control over whether a given driver supports the **clone open**. The decision to implement a STREAMS driver as a cloneable device is made by the designers of the device driver.

3 Multiplexed STREAMS

Multiplexer Configurations 3-1

Building a Multiplexer 3-4

Dismantling a Multiplexer 3-13

Routing Data Through a Multiplexer 3-14

Multiplexer Configurations

In the earlier chapters, Streams were described as linear connections of modules, where each invocation of a module is connected to, at most, one upstream module and one downstream module. While this configuration is suitable for many applications, others require 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 one that multiplexes data from several upper Streams over a single lower Stream, as shown in Figure 3-1. An upper Stream is one that is upstream from a multiplexer, and a lower Stream is one that is downstream from a multiplexer. A terminal windowing facility might be implemented in this fashion, where each upper Stream is associated with a separate window.

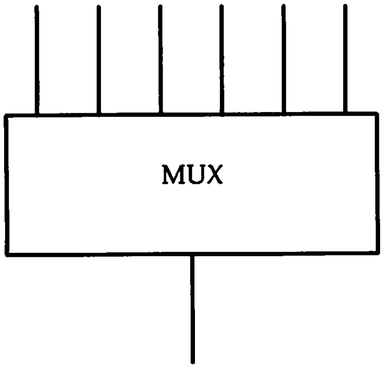

Figure 3-1: Many-to-One Multiplexer

A second type of multiplexer might route data from a single upper Stream to one of several lower Streams, as shown in Figure 3-2. An internetworking protocol could take this form, where each lower Stream links the protocol to a different physical network.

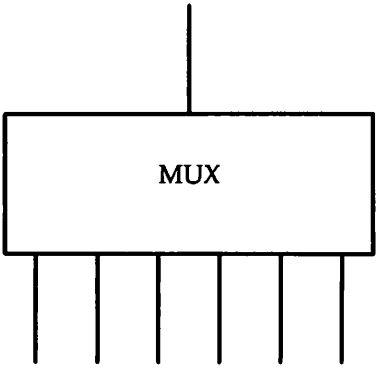

Figure 3-2: One-to-Many Multiplexer

A third type of multiplexer might route data from one of many upper Streams to one of many lower Streams, as shown in Figure 3-3.

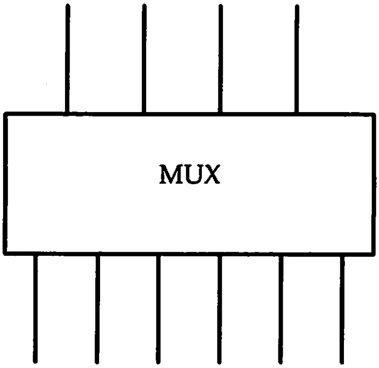

Figure 3-3: Many-to-Many Multiplexer

A STREAMS mechanism is available that supports the multiplexing of Streams through special pseudo-device drivers. Using a linking facility, users can dynamically build, maintain, and dismantle each of the above multiplexed Stream configurations. In fact, these configurations can be further combined to form complex, multilevel, multiplexed Stream configurations.

The remainder of this chapter describes multiplexed Stream configurations in the context of an example (see Figure 3-4). In this example, an internetworking protocol pseudo-device driver (IP) is used to route data from a single upper Stream to one of two lower Streams. This driver supports two STREAMS connections beneath it to two distinct sub-networks. One sub-network supports the IEEE 802.3 standard for the CSMA/CD medium access method. The second sub-network supports the IEEE 802.4 standard for the token-passing bus medium access method.

The example also presents a transport protocol pseudo-device driver (TP) that multiplexes multiple virtual circuits (upper Streams) over a single Stream to the IP pseudo-device driver.

Building a Multiplexer

Figure 3-4 shows the multiplexing configuration to be created. This configuration will enable users to access the services of the transport protocol. To free users from the need to know about the underlying protocol structure, a user-level daemon process will build and maintain the multiplexing configuration. Users can then access the transport protocol directly by opening the TP driver device node.

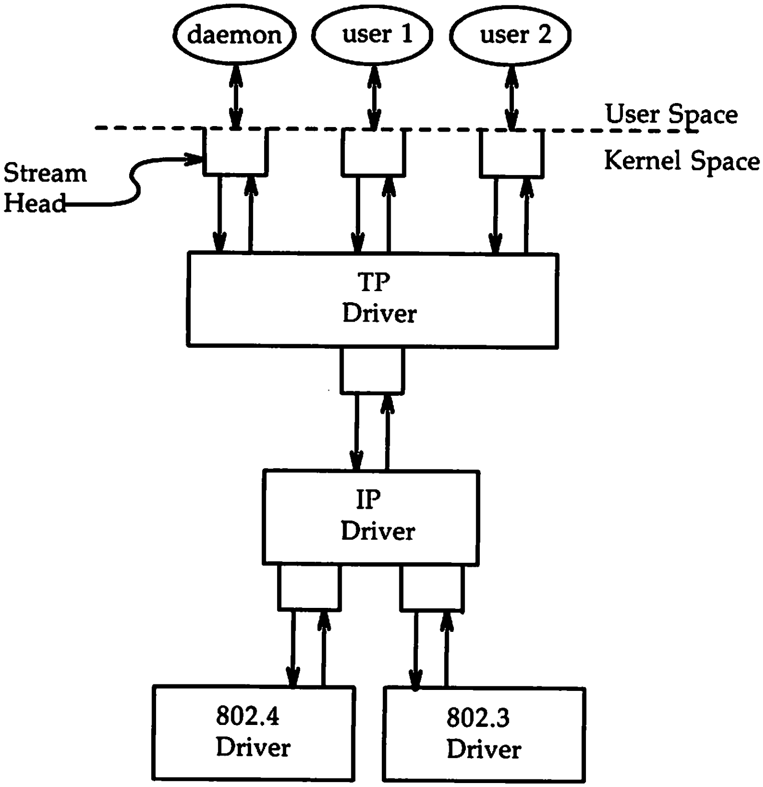
daemon
user 1
user 2
User Space
Kernel Space
Stream Head
TP Driver
IP Driver
802.4 Driver
802.3 Driver

Figure 3-4: Protocol Multiplexer

The following example shows how this daemon process sets up the protocol multiplexer. The necessary declarations and initialization for the daemon program are as follows:

```
#include <fcntl.h>
#include <stropts.h>

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);
   }
   setpgrp( );
```

This multilevel, multiplexed Stream configuration will be built from the bottom up. Therefore, the example begins by constructing the IP multiplexer. This multiplexing pseudo-device driver is treated like any other software driver. It owns a node in the UNIX 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, creating separate Streams above each driver as shown in Figure 3-5. The Stream to the 802.4 driver may now be connected below the multiplexing IP driver using the I_LINK **ioctl** call.

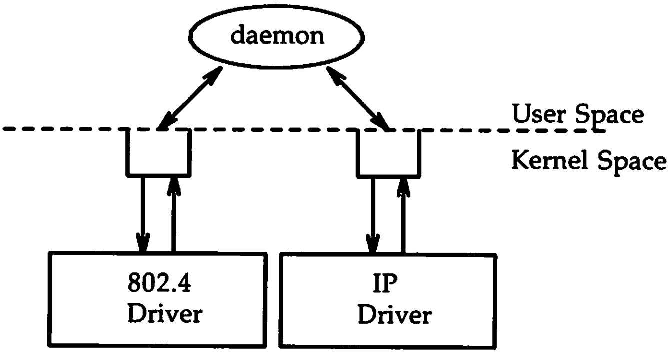
daemon
User Space
Kernel Space
802.4
Driver
IP
Driver

Figure 3-5: 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*, must reference the Stream connected to the multiplexing driver, and the second file descriptor, *fd_802_4*, must reference the Stream to be connected below the multiplexer. Figure 3-6 shows the state of these Streams following the I_LINK call. The complete Stream to the 802.4 driver has been connected below the IP driver, including the Stream head. The Stream head of the 802.4 driver will be used by the IP driver to manage the multiplexer.

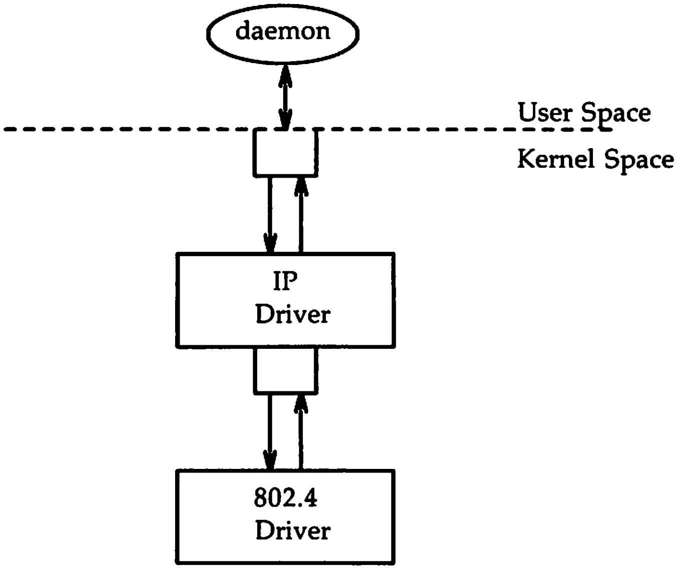

Figure 3-6: IP Multiplexer After First Link

I_LINK will return an integer value, called a mux ID, which is used by the multiplexing driver to identify the Stream just connected below it. This mux ID is ignored in the example, but may be useful for dismantling a multiplexer or routing data through the multiplexer. Its significance is discussed later.

The following sequence of system calls is used to continue building the internetworking 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);
}
```

All links below the IP driver have now been established, giving the configuration in Figure 3-7.

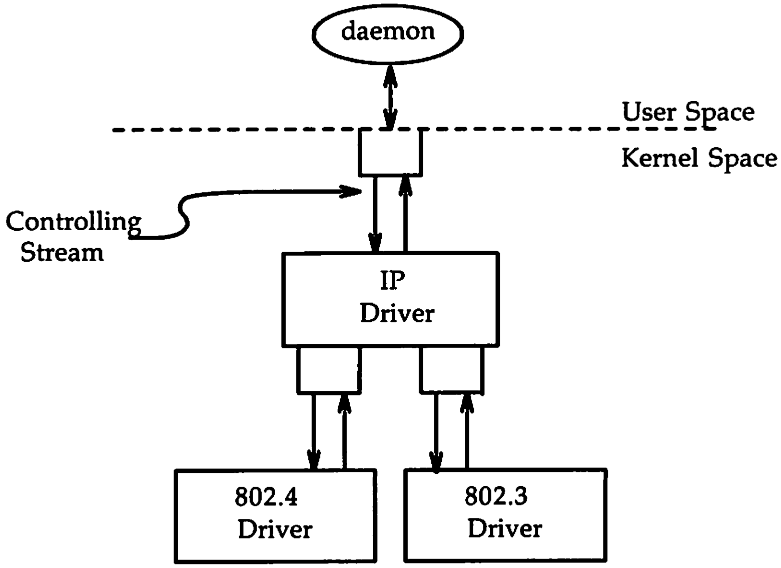
daemon
User Space
Kernel Space
Controlling Stream
IP Driver
802.4 Driver
802.3 Driver

Figure 3-7: IP Multiplexer

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, as will be illustrated later in this chapter. The Stream referenced by *fd_ip* is the controlling Stream for the IP multiplexer.

NOTE

The order in which the Streams in the multiplexing configuration are opened is unimportant. If, however, 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 (typically found in section 7 of the *User's/System Administrator's Reference Manual*) should describe 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 will now be linked below the transport protocol (TP) multiplexing driver. As seen earlier in Figure 3-4, only one link will be 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);
}
```

The multilevel multiplexing configuration shown in Figure 3-8 has now been created.

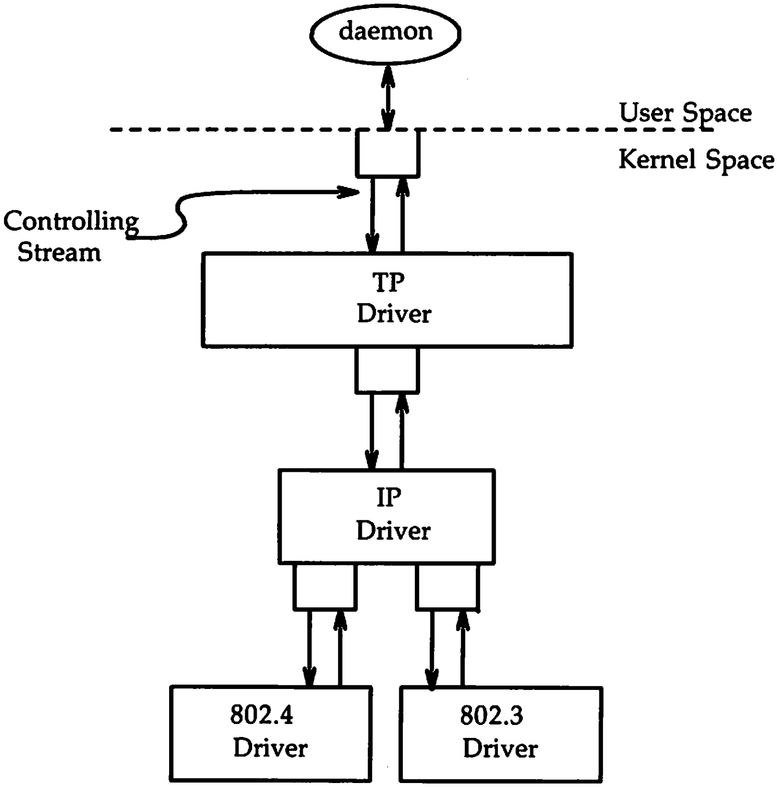
daemon
User Space
Kernel Space
Controlling Stream
TP Driver
IP Driver
802.4 Driver
802.3 Driver

Figure 3-8: TP Multiplexer

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. Closing these file descriptors may be necessary when building large multiplexers so that many devices can be linked together without exceeding the UNIX System limit on the number of simultaneously open files per process. If these file descriptors are not closed, all subsequent **read, write, ioctl, poll, getmsg,** and **putmsg** system calls issued to them will 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 will complete the multiplexing daemon example:

```
    close(fd_802_4);
    close(fd_802_3);
    close(fd_ip);

    /* Hold multiplexer open forever */
    pause();
}
```

Figure 3-4 shows the complete picture of the multilevel protocol multiplexer. The transport driver is designed to support several, simultaneous virtual circuits, where these virtual circuits map one-to-one to Streams opened to the transport driver. These Streams will be 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. Of course, the driver must be designed with the intelligence to route data from the single lower Stream to the appropriate upper Stream.

Notice in Figure 3-4 that 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 sub-networks that support the transport service.

Multilevel, multiplexing configurations, such as the one presented in the above example, should be assembled from the bottom up. That is because STREAMS does not allow **ioctl** requests (including I_LINK) to be passed through higher multiplexing drivers to reach the desired multiplexer; they must be sent directly to the intended driver. For example, once the IP driver is linked under the TP driver, **ioctl** requests cannot be sent to the IP driver through the TP driver.

Dismantling a Multiplexer

Streams connected to a multiplexing driver from above with **open** can be dismantled by closing each Stream with **close**. In the protocol multiplexer, these Streams correspond to the virtual circuit Streams above the TP multiplexer. The mechanism for dismantling Streams that have been linked below a multiplexing driver is less obvious and is described below in detail.

The I_UNLINK **ioctl** call is used to disconnect each multiplexer link below a multiplexing driver individually. This command takes the following form:

```
ioctl(fd, I_UNLINK, mux_id);
```

where *fd* is a file descriptor associated with a Stream connected to the multiplexing driver from above, and *mux_id* 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 *mux_id* value of -1 may be used to disconnect all drivers from the multiplexer simultaneously.

In the multiplexing daemon program presented earlier, 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 mux ID values returned by I_LINK could be ignored.

An important side effect of automatic dismantling on **close** is that it is not possible for a process to build a multiplexing configuration and then exit. That is because **exit** [see *exit*(2)] will close 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.

Routing Data Through a Multiplexer

As demonstrated, STREAMS has provided a mechanism for building multiplexed Stream configurations. However, the criteria on which a multiplexer routes data is driver-dependent. For example, the protocol multiplexer shown in the last example might use address information found in a protocol header to determine over which sub-network a given packet should be routed. It is the multiplexing driver's responsibility to define its routing criteria.

One routing option available to the multiplexer is to use the mux ID value to determine to which Stream data should be routed. (Remember that each multiplexer link is associated with a mux ID.) I_LINK passes the mux ID value to the driver and returns this value to the user. The driver can therefore specify that the mux ID value must 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 mux ID of the desired lower Stream into the first four bytes of each message passed to it. The driver could then match the mux ID in each message with the mux ID of each lower Stream and route the data accordingly.

4 Message Handling

Service Interface Messages 4-1
Service Interfaces 4-1

The Message Interface 4-4

Datagram Service Interface Example 4-7
Accessing the Datagram Provider 4-9
Closing the Service 4-13
Sending a Datagram 4-13
Receiving a Datagram 4-15

Service Interface Messages

A STREAMS message format has been defined to simplify the design of service interfaces. Also, two new system calls, **getmsg** and **putmsg**, are available for sending these messages downstream and receiving messages that are available at the Stream head. This chapter describes these system calls in the context of a service interface example. First, a brief overview of STREAMS service interfaces is presented.

Service Interfaces

A principal advantage of the STREAMS mechanism is its modularity. From user level, kernel-resident modules can be dynamically interconnected to implement any reasonable processing sequence. This modularity reflects the layering characteristics of contemporary network architectures.

One benefit of modularity is the ability to interchange modules of like function. For example, two distinct transport protocols, implemented as STREAMS modules, may provide a common set of services. An application or higher layer protocol that requires those services can use either module. This ability to substitute modules enables user programs and higher-level protocols to be independent of the underlying protocols and physical communication media.

Each STREAMS module provides a set of processing functions, or services, and an interface to those services. The service interface of a module defines the interaction between that module and any neighboring modules, and therefore is a necessary component for providing module substitution. By creating a well-defined service interface, applications and STREAMS modules can interact with any module that supports that interface. Figure 4-1 demonstrates this.

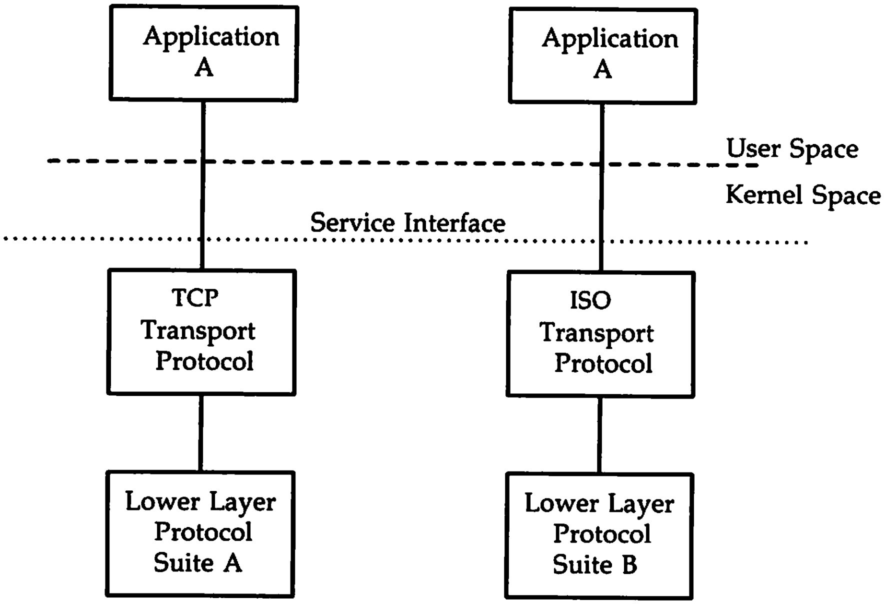
Application
A
Application
A
User Space
Kernel Space
Service Interface
TCP
Transport
Protocol
ISO
Transport
Protocol
Lower Layer
Protocol
Suite A
Lower Layer
Protocol
Suite B

Figure 4-1: Protocol Substitution

By defining a service interface through which applications interact with a transport protocol, it is possible to substitute a different protocol below that service interface in a manner completely transparent to the application. In this example, 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 4-2.

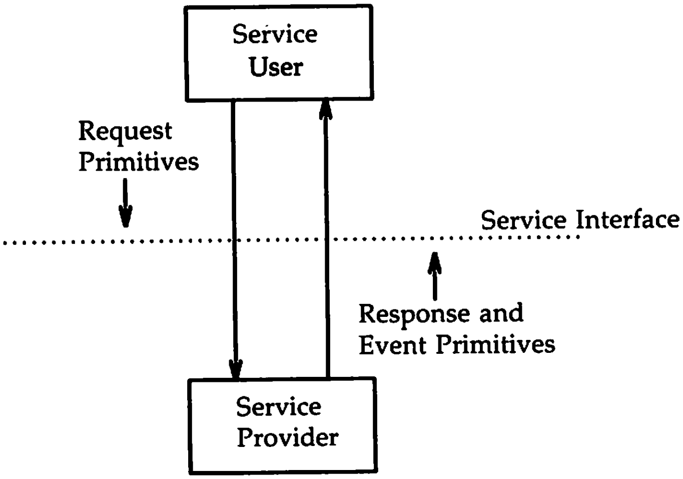
Service User
Request Primitives
Service Interface
Response and Event Primitives
Service Provider

Figure 4-2: Service Interface

Typically, a user makes a request 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. The service interface is defined as the set of primitives that define a service and the allowable state transitions that result as these primitives are passed between the user and provider.

The Message Interface

A message format has been defined to simplify the design of service interfaces using STREAMS. 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 may 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.

STREAMS enables modules to create these messages and pass them to neighbor modules. However, the **read** and **write** system calls are not sufficient to enable a user process to generate and receive such messages. First, **read** and **write** are byte-stream oriented, with no concept of message boundaries. To support service interfaces, the message boundary of each service primitive must be preserved so that the beginning and end of each primitive can be located. Also, **read** and **write** offer only one buffer to the user for transmitting and receiving STREAMS messages. If control information and data were placed in a single buffer, the user would have to parse the contents of the buffer to separate the data from the control information.

Two new STREAMS system calls are available that enable user processes to create STREAMS messages and send them to neighboring kernel modules and drivers or receive the contents of such messages from kernel modules and drivers. These system calls preserve message boundaries and provide separate buffers for the control and data parts of a message.

The **putmsg** system call enables a user to create STREAMS messages and send them downstream. The user supplies the contents of the control and data parts of the message in two separate buffers. Likewise, the **getmsg** system call retrieves such messages from a Stream and places the contents into two user buffers.

The syntax of **putmsg** is as follows:

```
int putmsg (fd, ctlptr, dataptr, flags)
int fd;
struct strbuf *ctlptr;
struct strbuf *dataptr;
int flags;
```

fd identifies the Stream to which the message will be passed, *ctlptr* and *dataptr* identify the control and data parts of the message, and *flags* may be used to specify that a priority message should be sent.

The **strbuf** structure is used to describe the control and data parts of a message and has the following format:

```
struct strbuf {
   int   maxlen;       /* maximum buffer length */
   int   len;          /* length of data */
   char  *buf;         /* pointer to buffer */
}
```

buf points to a buffer containing the data and *len* specifies the number of bytes of data in the buffer. *maxlen* specifies the maximum number of bytes the given buffer can hold and is only meaningful when retrieving information into the buffer using **getmsg**.

The **getmsg** system call retrieves messages available at the Stream head and has the following syntax:

```
int getmsg (fd, ctlptr, dataptr, flags)
int fd;
struct strbuf *ctlptr;
struct strbuf *dataptr;
int *flags;
```

The arguments to **getmsg** are the same as those for **putmsg**.

The remainder of this chapter presents an example that demonstrates how **putmsg** and **getmsg** may be used to interact with the service interface of a simple datagram protocol provider. A potential provider of such a service might be the IEEE 802.2 Logical Link Control Protocol Type 1. The example implements a user-level library that would free the user from knowledge of the underlying STREAMS system calls. The Transport Interface of the

Network Services Library in UNIX System Release 3.1 provides a similar function for transport layer services. The example here illustrates how a service interface might be defined, and is not an example of a complete IEEE 802.2 service interface.

Datagram Service Interface Example

The example datagram service interface library presented below includes four functions that enable a user to do the following:

- establish a Stream to the service provider and bind a protocol address to the Stream
- send a datagram to a remote user
- receive a datagram from a remote user
- close the Stream connected to the provider

First, the structure and constant definitions required by the library are shown. These typically will 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 */
};
```

continued

```
struct unitdata_req {        /* unitdata request */
   long  PRIM_type;          /* always UNITDATA_REQ */
   long  DEST_addr;          /* destination addr */
};

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

struct error_ack {           /* error acknowledgment */
   long  PRIM_type;          /* always ERROR_ACK */
   long  UNIX_error;         /* UNIX error 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 <stropts.h>
#include <stdio.h>
#include <errno.h>
```

Five primitives have been 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 a datagram 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. These are:

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 non-fatal 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 a datagram destined for the user has arrived.

The structures defined above 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.

Accessing the Datagram Provider

The first routine presented below, *inter_open*, opens the protocol driver device file specified by *path* and binds the protocol address contained in *addr* so that it may receive datagrams. 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.

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

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

    /* send bind request msg down stream */

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

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

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

After *inter_open* sends the bind request, it must wait for an acknowledgment from the service provider, as follows:

```
    /* 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:
            errno = EPROTO;
            close(fd);
            return(-1);

        case OK_ACK:
            return(fd);

        case ERROR_ACK:
            if (ctlbuf.len < sizeof(struct error_ack)) {
                errno = EPROTO;
                close(fd);
                return(-1);
            }
            error_ack = (struct error_ack *)&rcvbuf;
            errno = error_ack->UNIX_error;
            close(fd);
            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 *error_ack* structure, and no data part.

The acknowledgment primitives are defined as priority messages. Two classes of messages can arrive at the Stream head: priority and normal. Normal messages are queued in a first-in-first-out manner at the Stream head, while priority messages are placed at the front of the Stream head queue. The STREAMS mechanism allows only one priority message per Stream at the Stream head at one time; any further priority messages are discarded until the first message is processed. Priority messages are particularly suitable for acknowledging service requests when the acknowledgment should be placed ahead of any other messages at the Stream head.

NOTE These messages are not intended to support the expedited data capabilities of many communication protocols, as evidenced by the one-at-a-time restriction just described.

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 *flags* argument points to an integer containing the value **RS_HIPRI**. This flag indicates that **getmsg** should wait for a STREAMS priority message before returning and is set because the acknowledgment primitives are priority messages. Even if a normal message is available, **getmsg** will block until a priority message arrives.

On return from **getmsg**, the *len* field is checked 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

The next routine in the datagram service library is *inter_close,* which closes the Stream to the service provider.

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

The routine simply closes the given file descriptor. This will cause the protocol driver to free any resources associated with that Stream. For example, the driver may unbind the protocol address that had previously been bound to that Stream, thereby freeing that address for use by some other service user.

Sending a Datagram

The third routine, *inter_snd,* passes a datagram 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 and sets *errno* to an appropriate UNIX System error value.

```
inter_snd(fd, buf, len, addr)
char *buf;
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 datagram 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 datagram. The data to be transmitted is placed in the data part of the request message.

Unlike the bind request, the datagram request primitive requires no acknowledgment from the service provider. In the example, this choice was made to minimize the overhead during data transfer. Since datagram services are inherently unreliable, this is a valid design choice. If the **putmsg** call succeeds, this routine assumes all is well and returns the number of bytes passed to the service provider.

Receiving a Datagram

The final routine in this example, *inter_rcv,* retrieves the next available datagram. *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 datagram will be placed. On successful completion, *inter_rcv* returns the number of bytes in the retrieved datagram; on failure, it returns -1 and sets the appropriate UNIX System error value.

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

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

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

getmsg is called to retrieve the datagram 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 datagram sender. The data part contains the data itself.

In *ctlbuf, buf* must point to a buffer where the control information will be stored, and *maxlen* must be set to indicate the maximum size of that buffer. Similar initialization is done for *databuf.*

The *flags* argument to **getmsg** is set to zero, indicating that the next message should be retrieved from the Stream head, regardless of its priority. Datagrams will arrive in normal priority messages. If no message currently exists at the Stream head, **getmsg** will block until a message arrives.

The user's control and data buffers should be large enough to hold any incoming datagram. If both buffers are large enough, **getmsg** will process the datagram indication and return 0, indicating that a full message was retrieved successfully. However, if either buffer is not large enough, **getmsg** will only retrieve the part of the message that fits into each user buffer. The remainder of the message is saved for subsequent retrieval, and a positive, non-zero 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, non-zero value), the function will set *errno* to EIO and fail.

The type of the primitive returned by **getmsg** is checked to make sure it is a datagram indication. The source address is then set and the number of bytes of data in the datagram is returned.

The above example presents a simplified service interface. The state transition rules for such an interface were not presented for the sake of brevity. The intent was to show typical uses of the **putmsg** and **getmsg** system calls. See *putmsg*(2) and *getmsg*(2) for further details.

Introduction to Part 2

Part 2 of this guide, Module and Driver Programming, describes the use of STREAMS kernel facilities for developing and installing modules and drivers. It is intended for system programmers with knowledge of UNIX System kernel programming, device driver development, and networking and other data communication facilities. Knowledge of the *STREAMS Primer* and the *Driver Design Guide* is assumed.

STREAMS provides module and driver developers with integral functions, a set of utility routines, and facilities that expedite design and implementation. The principle development facilities are listed below:

- Message storage management—to maintain STREAMS' own memory resources for message storage
- Flow control—to conserve STREAMS memory and processing resources
- Scheduling—to control the execution of service procedures
- Multiplexing—to switch data among multiple Streams
- Error and trace loggers—for debugging and administrative use

Part 2 is organized as follows:

- Chapter 5, Streams Mechanism, reviews the operation of STREAMS and describes how a Stream is constructed and dismantled.
- Chapter 6, Modules, describes the basic STREAMS data structures and the organization of a module.
- Chapter 7, Messages, introduces message blocks, read and write system calls, and the message storage pool.
- Chapter 8, Message Queues and Service Procedures, discusses put and service procedures, message queueing, and basic flow control.
- Chapter 9, Drivers, describes STREAMS driver organization and discusses typical driver processing.
- Chapter 10, Complete Driver, provides a full implementation of a driver and describes the clone mechanism.

- Chapter 11, Multiplexing, describes the multiplexing facility.
- Chapter 12, Service Interface, discusses service interfaces within a Stream and at the Stream/user boundary.
- Chapter 13, Advanced Topics, contains advanced topics including signals and Stream head options.
- Appendix A, Kernel Structures, summarizes kernel structures used by modules and drivers.
- Appendix B, Message Types, describes STREAMS message types.
- Appendix C, Utilities, specifies the STREAMS kernel utility routines.
- Appendix D, Design Guidelines, summarizes module and driver design guidelines.
- Appendix E, Configuring, describes how modules and drivers are configured into the UNIX System, tunable parameters and STREAMS system error messages.
- The Glossary defines terms unique to STREAMS.

5 STREAMS Mechanism

Overview 5-1

Stream Construction 5-3

Opening a Stream 5-5

Adding and Removing Modules 5-6

Closing 5-7

Overview

A Stream implements a connection within the kernel between a driver in kernel space and a process in user space. It provides a general character input/output (I/O) interface for user processes which is upwardly compatible with the interface of the preexisting character I/O facilities. A Stream is analogous to a shell pipeline except that data flow and processing are bidirectional to support concurrent input and output.

The components that form a Stream are the Stream head, driver, and optional modules (see Figure 1 in the Preface). A Stream is initially constructed as the result of a user process **open** system call referencing a STREAMS file. The call causes a kernel resident driver to be connected with a Stream head to form a Stream. Subsequent **ioctl** calls select kernel resident modules and cause them to be inserted in the Stream. A module represents intermediate processing on messages flowing between the Stream head and driver. A module can function as, for example, a communication protocol, line discipline, or data filter. STREAMS allows a user to connect a module with any other module. The user determines the module connection sequences that result in useful configurations.

A process can send and receive characters on a Stream using **write** and **read**, as on character files. When user data enters the Stream head or external data enters the driver, the data is placed into messages for transmission on the Stream. All data passed on a Stream is carried in messages, each having a defined message type identifying the message contents. Internal control and status information is transmitted among modules or between the Stream and user process as messages of certain types interleaved on the Stream. Modules and drivers can send certain message types to the Stream head to cause the generation of signals or errors to be received by the user process.

A module is comprised of two identical sets of data structures called QUEUEs. One QUEUE is for upstream processing and the other is for downstream processing. The processing performed by the two QUEUEs is generally independent so that a Stream operates in a full-duplex manner. The interface between modules is uniform and simple. Messages flow from module to module. A message from one module is passed to the single entry point of its neighboring module.

The last **close** system call dismantles the Stream and closes the file, semantically identical to character I/O drivers.

STREAMS supports implementation of user-level applications with extensions to the above general system calls and STREAMS specific system calls: **putmsg**, **getmsg**, **poll**, and a set of STREAMS generic **ioctl** functions.

Stream Construction

STREAMS constructs a Stream as a linked list of kernel resident data structures. In a STREAMS file, the **inode** points to the Stream **header** structure. The **header** is used by STREAMS kernel routines to perform operations on this Stream generally related to system calls. Figure 5-1 depicts the downstream (write) portion of a Stream (see Chapter 3 of the *Primer*) connected to the **header.** There is one **header** per Stream. From the **header** onward, a Stream is constructed of QUEUEs. The upstream (read) portion of the Stream (not shown in Figure 5-1) parallels the downstream portion in the opposite direction and terminates at the Stream **header** structure.

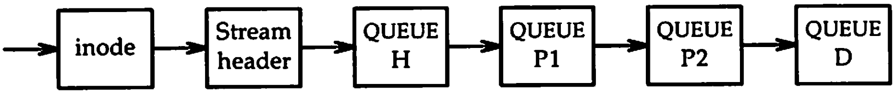

Figure 5-1: Downstream Stream Construction

At the same relative location in each QUEUE is the address of the entry point, a procedure to be executed on any message received by that QUEUE. The procedure for QUEUE H, at one end of the Stream, is the STREAMS-provided Stream head routine. QUEUE H is the downstream half of the Stream head. The procedure for QUEUE D, at the other end, is the driver routine. QUEUE D is the downstream half of the Stream end. P1 and P2 are pushable modules, each containing their own unique procedures. That is, all STREAMS components are of similar organization.

This similarity results in the uniform manner of navigating in either direction on a Stream: messages move from one end to the other, from QUEUE to the next linked QUEUE, executing the procedure specified in the QUEUE.

Figure 5-2 shows the data structures forming each QUEUE: **queue_t**, **qinit**, **module_info**, and **module_stat**. **queue_t** contains various modifiable values for this QUEUE, generally used by STREAMS. **qinit** contains a pointer to the processing procedures, **module_info** contains limit values and **module_stat** is used for statistics. The two QUEUEs in a module will generally each contain a different set of these structures. The contents of these structures are described in following chapters.

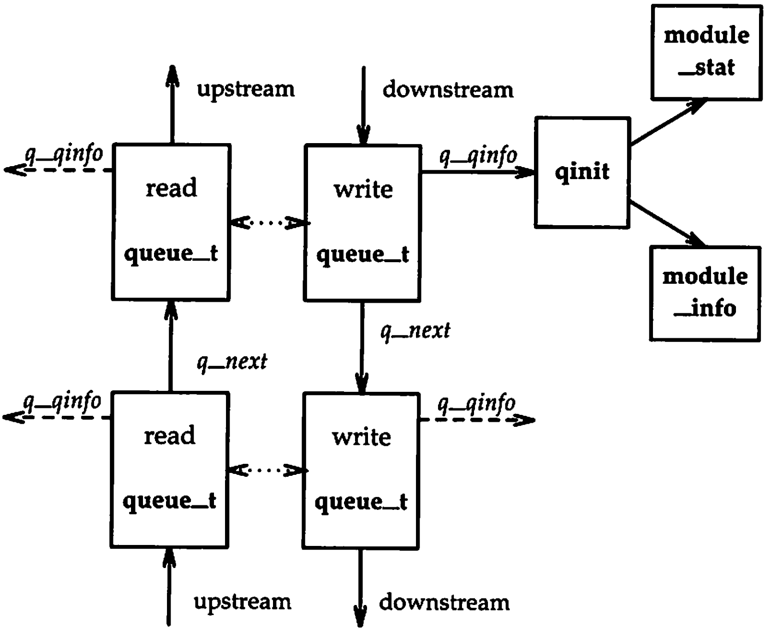
module
_stat
upstream
downstream
q_qinfo
read
queue_t
write
queue_t
q_qinfo
qinit
module
_info
q_next
q_next
q_qinfo
read
queue_t
write
queue_t
q_qinfo
upstream
downstream

Figure 5-2: QUEUE Data Structures

Figure 5-1 shows QUEUE linkage in one direction while Figure 5-2 shows two neighboring modules with links (solid vertical arrows) in both directions. When a module is pushed onto a Stream, STREAMS creates two QUEUEs and links each QUEUE in the module to its neighboring QUEUE in the upstream and downstream direction. The linkage allows each QUEUE to locate its next neighbor. The next relation is implemented between **queue_ts** in adjacent modules by the *q_next* pointer. Within a module, each **queue_t** locates its mate (see dotted arrows in Figure 5-2) by use of STREAMS macros, since there is no pointer between the two **queue_ts**. The existence of the Stream head and driver is known to the QUEUE procedures only as destinations towards which messages are sent.

Opening a Stream

When a file is opened [see *open*(2)], a STREAMS file is recognized by a non-null value in the *d_str* field of the associated **cdevsw** entry. *d_str* points to a **streamtab** structure:

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

streamtab defines a module or driver and points to the read and write **qinit** structures for the driver.

If this **open** call is the initial file open, a Stream is created. First, the single **header** structure and the Stream head (see Figure 5-1) **queue_t** structure pair are allocated. Their contents are initialized with predetermined values including, as noted above (see QUEUE H), the Stream head processing routines.

Then, a **queue_t** structure pair is allocated for the driver. The **queue_t** contents are zero unless specifically initialized (see Chapter 8). A single, common **qinit** structure pair is shared among all the Streams opened from the same **cdevsw** entry, as is the associated **module_info** and **module_stat** structures (see Figure 5-2).

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

If this **open** is not the initial open of this Stream, the only actions performed are to call the driver open and the open procedures of all pushable modules on the Stream.

Adding and Removing Modules

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

fmodsw is an array, analogous to **cdevsw**. Each **fmodsw** entry corresponds to a unique module and contains the name of the module (used by I_PUSH and certain other STREAMS **ioctls**) and a pointer to the module's **streamtab**. Next, STREAMS allocates **queue_t** structures and initializes their contents as in the driver open, above. As with the driver, the read and write **qinit** structures are shared among all the modules opened from this **fmodsw** entry (see Figure 5-2).

Then, *q_next* values are set and modified so that the module is interposed between the Stream head and the driver or module previously connected to the head. Finally, the module open procedure (located via **qinit**) is called. Unlike **open**, no other module or driver open procedure is called.

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

An **ioctl** I_POP [see *streamio*(7)] system call (pop) removes the module immediately below the Stream head. The pop calls the module close procedure. On return from the module close, any messages left on the module's message queues are freed (deallocated). Then, STREAMS connects the Stream head to the component previously below the popped module and deallocates the module's two **queue_t** structures. I_POP enables 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 or a new one inserted below a module. In the latter case, the original module is popped and pushed back after the new module has been pushed.

An I_POP cannot be used on a driver.

Closing

The last **close** system call to a STREAMS file dismantles the Stream. Dismantling consists of popping any modules on the Stream, closing the driver and closing the file. Before a module is popped by **close**, it may delay to allow any messages on the write message queue of the module to be drained by module processing. If O_NDELAY [see *open*(2)] is clear, **close** will wait up to 15 seconds for each module to drain. If O_NDELAY is set, the pop is performed immediately. **close** will also wait for the driver's write queue to drain. Messages can remain queued, for example, if flow control (see Chapter 6 in the *Primer*) is inhibiting execution of the write QUEUE. 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 message queues are freed, and the **queue_t** and **header** structures are deallocated.

NOTE

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

Finally, the file is closed.

6 Modules

Module Declarations 6-1

Module Procedures 6-4

Module and Driver Environment 6-6

Module Declarations

A module and driver will contain, as a minimum, declarations of the following form:

```
#include "sys/types.h"     /* required in all modules and drivers */
#include "sys/stream.h"    /* required in all modules and drivers */
#include "sys/param.h"

static struct module_info rminfo = { 0, "mod", 0, INFPSZ, 0, 0 };
static struct module_info wminfo = { 0, "mod", 0, INFPSZ, 0, 0 };
static int modopen( ), modrput( ), modwput( ), modclose( );

static struct qinit rinit = {
modrput, NULL, modopen, modclose, NULL, &rminfo, NULL
};
static struct qinit winit = {
modwput, NULL, NULL, NULL, NULL, &wminfo, NULL
};
struct streamtab modinfo = { &rinit, &winit, NULL, NULL };
```

The contents of these declarations are constructed for the null module example in this section. This module performs no processing; its only purpose is to show linkage of a module into the system. The descriptions in this section are general to all STREAMS modules and drivers unless they specifically reference the example.

The declarations shown are: the header set; the read and write QUEUE (*rminfo* and *wminfo*) **module_info** structures (see Figure 5-2); the module open, read-put, write-put and close procedures; the read and write (*rinit* and *winit*) **qinit** structures; and the **streamtab** structure.

The minimum header set for modules and drivers is **types.h** and **stream.h**. **param.h** contains definitions for NULL and other values for STREAMS modules and drivers as shown in the section titled "Accessible Symbols and Functions" in Appendix D.

NOTE Configuring a STREAMS module or driver (see Appendix E) does not require any procedures to be externally accessible, only **streamtab**. The **streamtab** structure name must be the prefix used in configuring, appended with "info."

As described in the previous chapter, **streamtab** contains **qinit** values for the read and write QUEUEs, pointing to a **module_info** and an optional **module_stat** structure. The two required structures, shown in Figure 5-2, are these:

```
struct qinit {
   int       (*qi_putp)( );       /* put procedure */
   int       (*qi_srvp)( );       /* service procedure */
   int       (*qi_qopen)( );      /* called on each open or a push */
   int       (*qi_qclose)( );     /* called on last close or a pop */
   int       (*qi_qadmin)( );     /* reserved for future use */
   struct module_info  *qi_minfo;     /* information structure */
   struct module_stat  *qi_mstat;     /* statistics structure - optional */
};

struct module_info {
   ushort     mi_idnum;     /* module ID number */
   char       *mi_idname;   /* module name */
   short      mi_minpsz;    /* min packet size accepted, for developer use */
   short      mi_maxpsz;    /* max packet size accepted, for developer use */
   short      mi_hiwat;     /* hi-water mark, for flow control */
   ushort     mi_lowat;     /* lo-water mark, for flow control */
};
```

qinit contains the QUEUE procedures. All modules and drivers with the same **streamtab** (i.e., the same **fmodsw** or **cdevsw** entry) point to the same upstream and downstream **qinit** structure(s). The structure is meant to be software read-only, as any changes to it affect all occurrences of that module in all Streams. Pointers to the open and close procedures must be contained in the read **qinit**. These fields are ignored in the write side. The example has no service procedure on the read or write side.

module_info contains identification and limit values. All modules and drivers with the same **streamtab** point to the same upstream and downstream **module_info** structure(s). As with **qinit**, this structure is intended to be software read-only. However, the four limit values are copied to **queue_t** (see Chapter 8) where they are modifiable. In the example, the flow control high- and low-water marks (see Chapter 9) are zero, since there are no service procedures, and messages are not queued in the module.

Three names are associated with a module: the character string in **fmodsw**, obtained from the name of the **/etc/conf/modules** directory used to configure the module (see Appendix E); the prefix for **streamtab**, used in configuring the module; and the module name field in the **module_info** structure. This field is a hook for future expansion and is not currently used. However, it is recommended that it be the same as the **module** name. The module name value used in the I_PUSH or other STREAMS **ioctl** commands is contained in **fmodsw**. Each module ID and module name should be unique in the system. The module ID is currently used only in logging and tracing (see Chapter 6 in the *Primer*). For the example in this chapter, the module ID is zero.

Minimum and maximum packet size are intended to limit the total number of characters contained in all (if any) of the M_DATA blocks in each message passed to this QUEUE. These limits are advisory except for the Stream head. For certain system calls that write to a Stream, the Stream head will observe the packet sizes set in the write QUEUE of the module immediately below it. Otherwise, the use of packet size is developer-dependent. In the example, INFPSZ indicates unlimited size on the read (input) side.

module_stat is optional, intended for future use. Currently, there is no STREAMS support for statistical information gathering. The structure is described in Appendix A.

Module Procedures

The null module procedures are as follows:

```
static int modopen(q, dev, flag, sflag)
    queue_t  *q;     /* pointer to read queue */
    dev_t     dev;   /* major/minor device number -- zero for modules */
    int       flag;  /* file open flags -- zero for modules */
    int       sflag; /* stream open flags */
{
    /* return success */
    return 0;
}
static int modwput(q, mp)/* write put procedure */
    queue_t  *q;     /* pointer to the write queue */
    mblk_t   *mp;    /* message pointer */
{
    putnext(q, mp);  /* pass message through */
}
static int modrput(q, mp)/* read put procedure */
    queue_t  *q;     /* pointer to the read queue */
    mblk_t   *mp;    /* message pointer */
{
    putnext(q, mp);  /* pass message through */
}
static int modclose(q, flag)
    queue_t  *q;     /* pointer to the read queue */
    int       flag;  /* file open flags - zero for modules */
{
}
```

The form and arguments of these four procedures are the same in all modules and all drivers. Modules and drivers can be used in multiple Streams and their procedures must be re-entrant.

modopen illustrates the open call arguments and return value. The arguments are the read queue pointer (*q*), the major/minor device number (*dev* in drivers only), the file open flags (*flag* is defined in **sys/file.h**), and the Stream open flag (*sflag*). For a module, the values of *flag* and *dev* are always zero. The Stream open flag can take on the following values:

MODOPEN	normal module open
0	normal driver open (see Chapter 9)
CLONEOPEN	clone driver open (see Chapter 10)

The return value from open is >= 0 for success and OPENFAIL for error. The open procedure is called on the first I_PUSH and on all subsequent **open** calls to the same Stream. During a push, a return value of OPENFAIL causes the I_PUSH to fail and the module to be removed from the Stream. If OPENFAIL is returned by a module during an **open** call, the **open** fails, but the Stream remains intact. For example, it can be returned by a module/driver that only wishes to be opened by a super-user:

```
        if (!suser( )) return OPENFAIL;
```

In the example, *modopen* simply returns successfully. *modrput* and *modwput* illustrate the common interface to put procedures. The arguments are the read or write **queue_t** pointer, as appropriate, and the message pointer. The put procedure in the appropriate side of the QUEUE is called when a message is passed from upstream or downstream. The put procedure has no return value. In the example, no message processing is performed. All messages are forwarded using the **putnext** macro (see Appendix C). **putnext** calls the put procedure of the next QUEUE in the proper direction.

The close procedure is only called on an I_POP or on the last **close** call of the Stream (see the last two sections of Chapter 5). The arguments are the read **queue_t** pointer and the file open flags as in *modopen*. For a module, the value of *flag* is always zero. There is no return value. In the example, *modclose* does nothing.

Module and Driver Environment

As discussed in Chapter 7 of the *Primer,* user context is not generally available to STREAMS module procedures and drivers. The exception is during execution of the open and close routines. Driver and module open and close routines have user context and may access the **u_area** structure (defined in **user.h**, see "Accessible Symbols and Functions" in Appendix D). These routines are allowed to sleep, but must always return to the caller. That is, if they sleep, it must be at priority <= PZERO, or with PCATCH set in the sleep priority. A process that is sleeping at priority > PZERO and is sent a signal via **kill** never returns from the sleep call. Instead, the system call is aborted.

CAUTION

STREAMS driver and module put procedures and service procedures have no user context. They cannot access the **u_area** structure of a process and must not sleep.

7 Messages

Message Format 7-1
Message Generation and Reception 7-3

Filter Module Declarations 7-5
bappend Subroutine 7-6

Message Allocation 7-8

Put Procedure 7-9

Message Format

Messages are the means of communication within a Stream. A message contains data or information identified by one of 18 message types (see Appendix B). Messages may be generated by a driver, a module, or the Stream head. The contents of certain message types can be transferred between a process and a Stream by use of system calls. STREAMS maintains its own pools for allocation of message storage.

All messages are composed of one or more message blocks. A message block is a linked triplet, two structures, and a variable-length buffer block. The structures are *msgb* (**mblk_t**), the message block, and *datab* (**dblk_t**), the data block:

```
struct msgb {
   struct      msgb      *b_next;/* next message on queue */
   struct      msgb      *b_prev;/* previous message on queue */
   struct      msgb      *b_cont;/* next message block of message */
   unsigned    char      *b_rptr;/* first unread byte in buffer */
   unsigned    char      *b_wptr;/* first unwritten byte in buffer  */
   struct      datab     *b_datap;/* data block */
};
typedef struct msgb mblk_t;

struct datab {
   struct      datab     *db_freep;/* used internally */
   unsigned    char      *db_base;/* first byte of buffer * */
   unsigned    char      *db_lim;/* last byte+1 of buffer */
   unsigned    char       db_ref;/* count of messages pointing to this block */
   unsigned    char       db_type;/* message type */
   unsigned    char       db_class;/* used internally */
};
typedef struct datab dblk_t;
```

mblk_t is used to link messages on a message queue, link the blocks in a message, and manage the reading and writing of the associated buffer. *b_rptr* and *b_wptr* are used to locate the data currently contained in the buffer. As shown in Figure 7-1, **mblk_t** points to the data block of the triplet. The data block contains the message type, buffer limits, and control variables. STREAMS allocates message buffer blocks of varying sizes (see below). *db_base* and *db_lim* are the fixed beginning and end (+1) of the buffer.

A message consists of one or more linked message blocks. Multiple message blocks in a message can occur, for example, because of buffer size limitations, or as the result of processing that expands the message. When a message is composed of multiple message blocks, the type associated with the first message block determines the message type, regardless of the types of the attached message blocks.

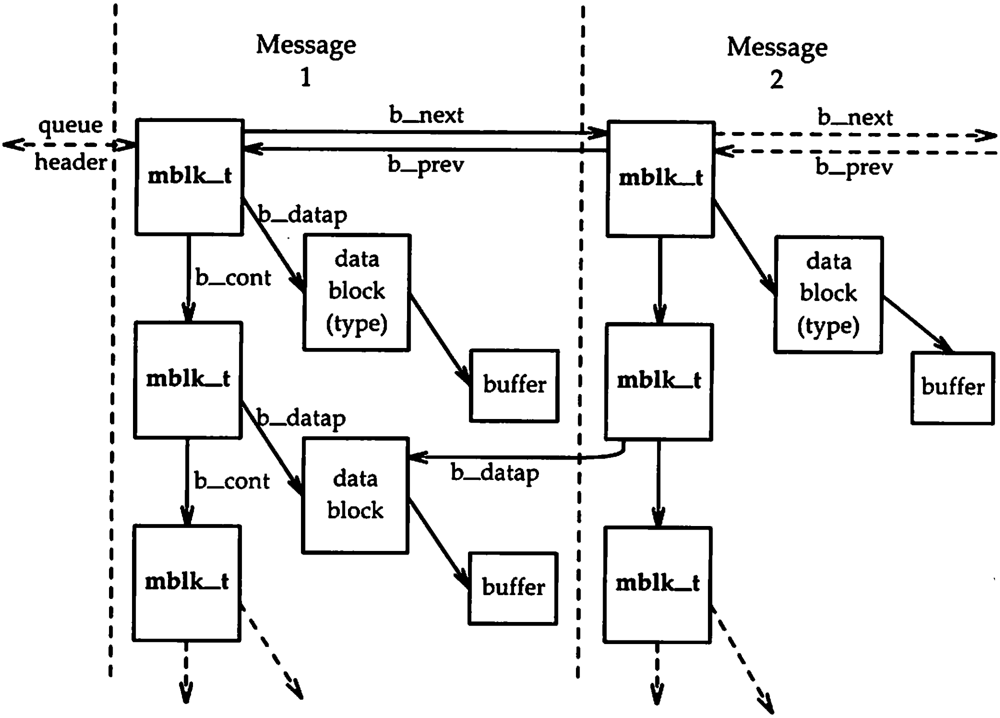
Message
1
Message
2
queue
header
b_next
b_prev
b_next
b_prev
mblk_t
mblk_t
b_datap
b_datap
b_cont
b_cont
data
block
(type)
data
block
(type)
buffer
buffer
mblk_t
mblk_t
b_datap
data
block
b_datap
buffer
mblk_t
mblk_t

Figure 7-1: Message Form and Linkage

A message may occur singly, as when it is processed by a put procedure, or it may be linked on the message queue in a QUEUE, generally waiting to be processed by the service procedure. Message 1, as shown in Figure 7-1, links to message 2. In the first message on a queue, *b_prev* points back to the header in the QUEUE. The last *b_next* points to the tail.

Note that a data block in message 1 is shared between message 1 and another message. Multiple message blocks can point to the same data block to conserve storage and to avoid copying overhead. For example, the same data block, with associated buffer, may be referenced in two messages, from separate modules that implement separate protocol levels. (Figure 7-1 illustrates the concept, but data blocks would not typically be shared by messages on the same queue.) The buffer can be retransmitted, if required by errors or timeouts, from either protocol level without replicating the data. Data block sharing is accomplished by means of a utility routine (see **dupmsg** in Appendix C). STREAMS maintains a count of the message blocks sharing a data block in the *db_ref* field.

STREAMS provides utility routines and macros, specified in Appendix C, to assist in managing messages and message queues, and to assist in other areas of module and driver development. A utility should always be used when operating on a message queue or accessing the message storage pool.

Message Generation and Reception

As discussed in the "Message Types" section in Chapter 4 of the *Primer*, most message types can be generated by modules and drivers. A few are reserved for the Stream head. The most commonly used types are M_DATA, M_PROTO, and M_PCPROTO. These, and certain other message types, can also be passed between a process and the topmost module in a Stream, with the same message boundary alignment maintained on both sides of the kernel. This allows a user process to function, to some degree, as a module above the Stream and maintain a service interface (see Chapter 12). M_PROTO and M_PCPROTO messages are intended to carry service interface information among modules, drivers, and user processes. Some message types can only be used within a Stream and cannot be sent or received from user level.

As discussed previously, modules and drivers do not interact directly with any system calls except **open** and **close**. The Stream head handles all message translation and passing. Message transfer between process and Stream head can occur in different forms. For example, M_DATA, M_PROTO, or M_PCPROTO messages can be transferred in their direct form by **getmsg** and

putmsg system calls (see Chapter 12). Alternatively, a **write** causes one or more M_DATA messages to be created from the data buffer supplied in the call. M_DATA messages received from downstream at the Stream head will be consumed by **read** and copied into the user buffer. As another example, M_SIG causes the Stream head to send a signal to a process (see Chapter 13).

Any module or driver can send any message type in either direction on a Stream. However, based on their intended use in STREAMS and their treatment by the Stream head, certain message types can be categorized as upstream, downstream or bidirectional. M_DATA, M_PROTO, or M_PCPROTO messages, for example, can be sent in both directions. Other message types are intended to be sent upstream to be processed only by the Stream head. Downstream messages are silently discarded if received by the Stream head.

Filter Module Declarations

The module shown below, *crmod,* is an asymmetric filter. On the write side, newline is converted to carriage return followed by newline. On the read side, no conversion is done. The declarations are essentially the same as the null module of the preceding chapter:

```
/* Simple filter - converts newline -> carriage return, newline */

#include "sys/types.h"
#include "sys/param.h"
#include "sys/stream.h"

static struct module_info minfo = { 0, "crmod", 0, INFPSZ, 0, 0 };

static int modopen(), modrput(), modwput(), modclose();
static struct qinit rinit = {
   modrput, NULL, modopen, modclose, NULL, &minfo, NULL
};
static struct qinit winit = {
   modwput, NULL, NULL, NULL, NULL, &minfo, NULL
};
struct streamtab crmdinfo = { &rinit, &winit, NULL, NULL };
```

Note that, in contrast to the null module example, a single **module_info** structure is shared by the read and write sides. A **config** file to configure *crmod* is shown in Appendix E.

modopen, modrput, and *modclose,* are the same as in the null module of the preceding chapter.

bappend Subroutine

The module makes use of a subroutine, *bappend,* which appends a character to a message block:

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

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

The *bappend* subroutine receives a pointer to a message block pointer 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**, described below.

If the **allocb** fails, *bappend* returns success, silently discarding the character. This may or may not be acceptable. For TTY-type devices, it is generally accepted. If the original message block is not full or the **allocb** is successful, *bappend* stores the character in the block.

Message Allocation

The **allocb** utility (see Appendix C) is used to allocate message storage from the STREAMS pool. Its declaration is:

```
mblk_t *allocb(buffersize, priority).
```

allocb will return a message block containing a buffer of at least the size requested, providing there is a buffer available at the message pool priority specified, or it will return NULL on failure. Three levels of message pool priority can be specified (see Appendix C). Priority generally does not affect **allocb** until the pool approaches depletion. In this case, for the same internal level of pool resources, **allocb** will fail low priority requests while granting higher priority requests. This allows module and driver developers to use STREAMS memory resources to their best advantage and for the common good of the system. Message pool priority does not affect subsequent handling of the message by STREAMS. BPRI_HI is intended for special situations. This transmission of urgent messages relates to time-sensitive events, conditions that could result in loss of state, loss of data, or inability to recover. BPRI_MED might be used, for example, when requesting an M_DATA buffer for holding input, and BPRI_LO might be used for an output buffer (presuming the output data can wait in user space). The Stream head uses BPRI_LO to allocate messages to contain output from a process (e.g., by **write** or **putmsg**). Note that **allocb** will always return a message of type M_DATA. The type may then be changed if required. *b_rptr* and *b_wptr* are set to *db_base* (see **mblk_t** and **dblk_t**).

allocb may return a buffer larger than the size requested. In *bappend,* if the message block contents were intended to be limited to MODBLKSZ, a check would have to be inserted.

If **allocb** indicates buffers are not available, the **bufcall** utility can be used to defer processing in the module or the driver until a buffer becomes available (**bufcall** is described in Chapter 13).

Put Procedure

The *modwput* function processes all the message blocks in any downstream data (type M_DATA) messages.

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

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

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

                    bp->b_rptr++;
                    continue;

              newblk:
                    if (nmp == NULL)
                          nmp = nbp;
                    else linkb(nmp, nbp); /* link message block to tail of nmp */
                    nbp = NULL;
              }
        }
        if (nmp == NULL)
              nmp = nbp;
        else linkb(nmp, nbp);
        freemsg(mp); /* deallocate message */
```

continued

```
                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 non-NULL, the new block is linked to the end of *nmp* by use of the **linkb** utility.

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**. If a new message exists, it is sent downstream.

8

Message Queues and Service Procedures

The queue_t Structure 8-1

Service Procedures 8-2

Message Queues and Message Priority 8-4

Flow Control 8-6

Example 8-8

Procedures 8-8

The queue_t Structure

Service procedures, message queues and priority, and basic flow control are all intertwined in STREAMS. A QUEUE will generally not use its message queue if there is no service procedure in the QUEUE. The function of a service procedure is to process messages on its queue. Message priority and flow control are associated with message queues.

The operation of a QUEUE revolves around the **queue_t** structure:

```
struct queue {
   struct qinit *q_qinfo;  /* procedures and limits for queue */
   struct msgb  *q_first;  /* head of message queue for this QUEUE */
   struct msgb  *q_last;   /* tail of message queue for this QUEUE */
   struct queue *q_next;   /* next QUEUE in Stream*/
   struct queue *q_link;   /* link to next QUEUE on STREAMS scheduling queue */
   caddr_t       q_ptr;    /* to private data structure */
   ushort        q_count;  /* weighted count of characters on message queue */
   ushort        q_flag;   /* QUEUE state */
   short         q_minpsz; /* min packet size accepted by this QUEUE */
   short         q_maxpsz; /* max packet size accepted by this QUEUE */
   ushort        q_hiwat;  /* message queue high-water mark, for flow control */
   ushort        q_lowat;  /* message queue low-water mark, for flow control */
};
typedef struct queue queue_t;
```

As described previously, two of these structures form a module. When a **queue_t** pair is allocated, their contents are zero unless specifically initialized. The following fields are initialized by STREAMS:

- *q_qinfo* - from **streamtab**
- *q_minpsz, q_maxpsz, q_hiwat, q_lowat* - from **module_info**

Copying values from **module_info** allows them to be changed in the **queue_t** without modifying the template (e.g., **streamtab** and **module_info**) values.

q_count is used in flow control calculations and is the weighed sum of the sizes of the buffer blocks currently on the message queue. The actual number of bytes in the buffer is not used. This is done to encourage the use of the smallest buffer that will hold the data intended to be placed in the buffer.

Service Procedures

Put procedures are generally required in pushable modules. Service procedures are optional. The general processing flow when both procedures are present is as follows: A message is received by the put procedure in a QUEUE, where some processing may be performed on the message. The put procedure transfers the message to the service procedure by use of the **putq** utility. **putq** places the message on the tail (see *q_last* in **queue_t**) of the message queue. Then, **putq** will generally schedule (using *q_link* in **queue_t**) the QUEUE for execution by the STREAMS scheduler following all other QUEUEs currently scheduled. After some indeterminate delay (intended to be short), the scheduler calls the service procedure. The service procedure gets the first message (*q_first*) from the message queue with the **getq** utility. The service procedure processes the message and passes it to the put procedure of the next QUEUE with **putnext**. The service procedure gets the next message and processes it. This FIFO processing continues until the queue is empty or flow control blocks further processing. The service procedure returns to caller.

CAUTION

A service routine must never sleep and 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** can be placed in the **qinit** structure declaration for the appropriate QUEUE side, to queue the message for the service procedure, e.g.,

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

More typically, put procedures will, as a minimum, process priority messages (see below) to avoid queueing them.

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. The presence of a service procedure is mandatory if the flow control mechanism is to be utilized by the QUEUE.

The delay for STREAMS to call a service procedure will vary with implementation and system activity. However, once the service procedure is scheduled, it is guaranteed to be called before user-level activity is resumed.

See also the section titled "Put and Service Procedures" in Chapter 5 of the *Primer.*

Message Queues and Message Priority

Figure 8-1 depicts a message queue linked by *b_next* and *b_prev* pointers. As discussed in the *Primer,* message queues grow when the STREAMS scheduler is delayed from calling a service procedure because of system activity, or when the procedure is blocked by flow control. When it is called by the scheduler, the service procedure processes enqueued messages in FIFO order. However, certain conditions require that the associated message (e.g., an M_ERROR) reach its Stream destination as rapidly as possible. STREAMS does this by assigning all message types to one of the two levels of message queueing priority—priority and ordinary. As shown in Figure 8-1, when a message is queued, the **putq** utility places priority messages at the head of the message queue in a FIFO order of queueing.

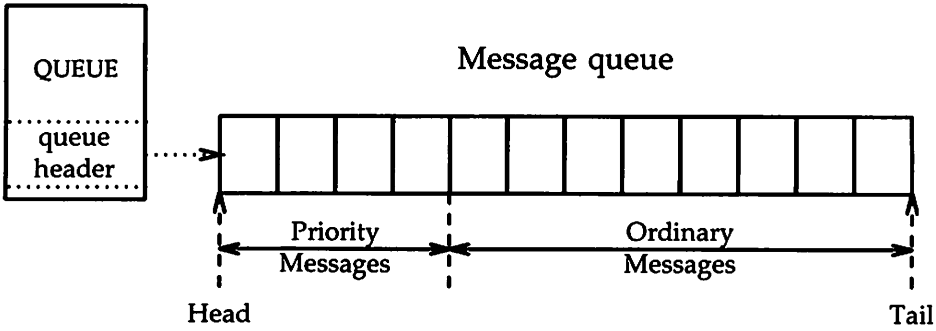

Figure 8-1: Message Queue Priority

Priority messages are not subject to flow control. When they are queued by **putq**, the associated QUEUE is always scheduled (in the same manner as any QUEUE; following all other QUEUEs currently scheduled). When the service procedure is called by the scheduler, the procedure uses **getq** to retrieve the first message on queue, which will be a priority message, if present. Service procedures must be implemented to act on priority messages immediately (see next section). The above mechanisms—priority message queueing, absence of flow control and immediate processing by a procedure—result in rapid transport of priority messages between the originating and destination components in the Stream.

The priority level for each message type is shown in Appendix B. Message queue management utilities are provided for use in service procedures (see Appendix C).

Flow Control

The elements of flow control are discussed in Chapter 6 of the *Primer*. Flow control is only used in a service procedure. Module and driver coding should observe the following guidelines for message priority. Priority messages, determined by the type of the first block in the message,

```
(bp->b_datap->db_type > QPCTL),
```

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

For ordinary messages, flow control must be tested before any processing is performed. The **canput** utility determines if the forward path from the QUEUE is blocked by flow control. The manner in which STREAMS determines flow control status for modules and drivers is described under "Driver Flow Control" in Chapter 9.

This is the general processing for flow control: Retrieve the message at the head of the queue with **getq**. Determine if the type is priority and not to be processed here. If both are true, pass the message to the put procedure of the following QUEUE with **putnext**. If the type is ordinary, use **canput** to determine if messages can be sent onward. If **canput** indicates messages should not be forwarded, put the message back on the queue with **putbq** and return from the procedure. In all other cases, process the message.

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

```
while (getq != NULL)
    if (priority message || canput)
        process message
        putnext
    else
        putbq
        return
```

NOTE A service procedure must process all messages on its queue unless flow control prevents this.

When an ordinary message is enqueued by **putq**, **putq** will cause the service procedure to be scheduled only if the queue was previously empty. If there are messages on the queue, **putq** presumes the service procedure is blocked by flow control and the procedure will be 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 (e.g., no buffers), it must assure it will return later (e.g., by use of **bufcall**, see Chapter 13) or it must discard all messages on queue. If this is not done, STREAMS will never schedule the service procedure to be run unless the QUEUE's put procedure queues a priority message with **putq**.

putbq replaces messages at the beginning of the appropriate section of the message queue in accordance with their message type priority (see Figure 8-1). This might not be the same position at which the message was retrieved by the preceding **getq**. A subsequent **getq** might return a different message.

Example

The filter module example of Chapter 7 is modified to have a service procedure, as shown below. The declarations from the example in Chapter 7 are unchanged except for the following lines (changes are shown in **bold**):

```
#include "sys/stropts.h"

static struct module_info minfo = {
   0, "ps_crmod", 0, INFPSZ, 512, 128
};
static int modopen( ), modrput( ), modwput( ), modwsrv(), modclose( );

static struct qinit winit = {
   modwput, modwsrv, NULL, NULL, NULL, &minfo, NULL
};
```

stropts.h is generally intended for user level. However, it includes definitions of flush message options common to user level, modules and drivers. **module_info** now includes the flow control high- and low-water marks (512 and 128) for the write QUEUE. (Even 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.) **qinit** now contains the service procedure pointer. *modopen, modclose,* and *modrput* (read side put procedure) are unchanged from Chapters 6 and 7. The *bappend* subroutine is also unchanged from Chapter 7.

Procedures

The write side put procedures and the beginning of the service procedure are shown next:

```
static int modwput(q, mp)
queue_t *q;
register 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 */
}

static int modwsrv(q) queue_t *q; {
   mblk_t *mp;

   while ((mp = getq(q) != NULL) {
       switch (mp->b_datap->db_type) {

           default:
               /* always putnext priority messages */
               if (mp->b_datap->db_type > QPCTL || canput(q->q_next)) {
                   putnext(q, mp);
                   continue;
               }
               else {
                   putbq(q, mp);
                   return;
               }

           case M_FLUSH:
               if (*mp->b_rptr & FLUSHW)
                   flushq(q, FLUSHDATA);
               putnext(q, mp);
               continue;
```

ps_crmod performs a similar function to *crmod* of the previous chapter, but it uses a service routine.

modwput, the write put procedure, switches on the message type. Priority messages that are not type M_FLUSH are **putnext** to avoid scheduling. The others are queued for the service procedure. An M_FLUSH message is a request to remove all 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_t**. *modwsrv* processes only one priority message, M_FLUSH. All other priority messages are passed through. Actually, no other priority messages should reach *modwsrv.* The check is included to show the canonical form when priority messages are queued by the put procedure.

For an M_FLUSH message, *modwsrv* checks the first data byte. If FLUSHW (defined in **stropts.h**) is set in the byte, the write queue is flushed by use of **flushq**. **flushq** takes two arguments, the queue pointer and a flag. The flag indicates what should be flushed, data messages (FLUSHDATA) or everything (FLUSHALL). In this case, data includes M_DATA, M_PROTO, and M_PCPROTO messages. The choice of what types of messages to flush is module-specific. As a general rule, FLUSHDATA should be used.

Ordinary messages will be returned to the queue if

```
canput(q->q_next)
```

returns false, indicating the downstream path is blocked.

In the remaining part of *modwsrv,* M_DATA messages are processed similarly to the previous example:

```
            case M_DATA: {
                mblk_t *nbp = NULL;
                mblk_t *next;

                if (!canput(q->q_next)) {
                    putbq(q, mp);
                    return;
                }
                /* 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:
                        putnext(q, nbp);
                        nbp = NULL;
                        if (!canput(q->q_next)) {
                            if (mp->b_rptr >= mp->b_wptr) {
                                next = mp->b_cont;
                                freeb(mp);
                                mp=next;
                            }
                            if (mp)
                                putbq(q, mp);
                            return;
                        }
                    }
                    next = mp->b_cont;
                    freeb(mp);
                }
                if (nbp)
                    putnext(q, nbp);
            }
        }
    }
}
```

The differences in M_DATA processing between this and the previous example relate to the manner in which the new messages are forwarded and flow control. 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** rather than being linked into a single, large message (as was done in the previous example). This alternative may not be desirable because message boundaries will be 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 (**canput**) after, rather than before, 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.

Another difference between the two examples is that each message block of the original message is returned to the pool with **freeb** when its processing is completed.

9 Drivers

Overview of Drivers 9-1

Driver Flow Control 9-3

Driver Programming 9-4
Driver Declarations 9-4
Driver Open 9-6

Driver Processing Procedures 9-8
Driver Flush Handling 9-9
Driver Interrupt 9-10

Driver and Module Ioctls 9-12

Driver Close 9-15

Overview of Drivers

This chapter describes the organization of a STREAMS driver and discusses some of the processing typically required in drivers. Certain elements of driver flow control are discussed. Procedures for handling user ioctls, common to modules and drivers, are described.

As discussed under "Stream Construction" in Chapter 5, driver and module organization are very similar. The call interfaces to all the driver procedures are identical to module interfaces and driver procedures must be reentrant. As described under "Environment" in Chapter 6, the driver put and service procedures have no user environment and cannot sleep. Other than with **open** and **close**, a driver interfaces with a user process by messages, and indirectly, through flow control.

There are two significant differences between modules and drivers. First, a device driver must also be accessible from an interrupt as well as from the Stream, and second, a driver can have multiple Streams connected to it. Multiple connections occur when more than one minor device uses the same driver and in the case of multiplexers (see Chapter 11). However, these particular differences are not recognized by the STREAMS mechanism: They are handled by developer-provided code included in the driver procedures.

Figure 9-1 shows multiple Streams (corresponding to minor devices) to a common driver. This depiction of two Streams connected to a single driver (also used in the *Primer*) is somewhat misleading. These are really two distinct Streams opened from the same **cdevsw** (i.e., same major device). Consequently, they have the same **streamtab** and the same driver procedures. Modules opened from the same **fmodsw** might be depicted similarly if they had any reason to be cognizant, as do drivers, of common resources or alternate occurrences.

Multiple occurrences (minor devices) of the same driver are handled during the initial open for each device. Typically, the **queue_t** address is stored in a driver-private structure indexed by the minor device number. The structure is typically pointed at by *q_ptr* (see Chapter 8). When the messages are received by the QUEUE, the calls to the driver put and service procedures pass the address of the **queue_t**, allowing the procedures to determine the associated device.

In addition to these differences, a driver is always 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.

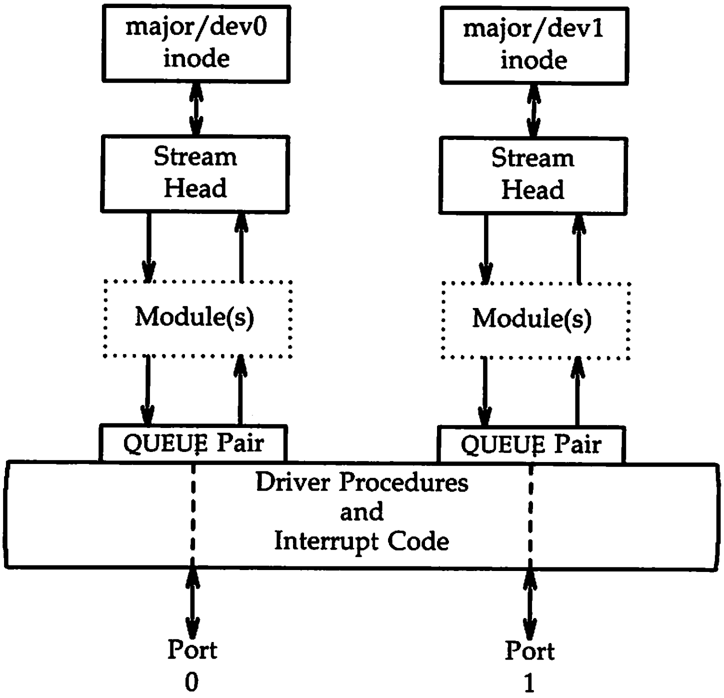
major/dev0
inode
major/dev1
inode
Stream
Head
Stream
Head
Module(s)
Module(s)
QUEUE Pair
QUEUE Pair
Driver Procedures
and
Interrupt Code
Port
0
Port
1

Figure 9-1: Device Driver Streams

Driver Flow Control

The same utilities (described in Chapter 8) and mechanisms used for module flow control are used by drivers. However, they are typically used in a different manner in drivers, because a driver generally does not have a service procedure. The developer sets flow control values (*mi_hiwat* and *mi_lowat*) in the write side **module_info** structure, which STREAMS will copy into *q_hiwat* and *q_lowat* in the **queue_t** structure of the QUEUE. A device driver typically has no write service procedure, but does maintain a write message queue. When a message is passed to the driver write side put procedure, the procedure will determine if device output is in progress. In the event output is busy, the put procedure cannot immediately send the message and calls the **putq** utility (see Appendix C) to queue the message. (Note that the driver might have elected to queue the message in all cases.) **putq** recognizes the absence of a service procedure and does not schedule the QUEUE.

When the message is queued, **putq** increments the value of *q_count* (approximately the enqueued character count, see the beginning of Chapter 8) by the size of the message and compares the result against the driver's write high-water limit (*q_hiwat*) value. If the count exceeds *q_hiwat*, **putq** will set the internal FULL (see the section titled "Flow Control" in Chapter 6 of the *Primer*) indicator for the driver write QUEUE. This will cause messages from upstream to be halted (**canput** returns FALSE) until the write queue count reaches *q_lowat*. The driver messages waiting to be output are dequeued by the driver output interrupt routine with **getq**, which decrements the count. If the resulting count is below *q_lowat*, **getq** will back-enable any upstream QUEUE that had been blocked. The above STREAMS processing also applies to modules on both write and read sides of the Stream.

Device drivers typically discard input when unable to send it to a user process. However, STREAMS allows flow control to be used on the driver read side, possibly to handle temporary upstream blocks. This is described in Chapter 13 in the section titled "Advanced Flow Control."

To some extent, a driver or module can control when its upstream transmission will become blocked. Control is available through the M_SETOPTS message (see Chapter 13 and Appendix B) to modify the Stream head read side flow control limits.

Driver Programming

The example below shows how a simple interrupt-per-character line printer driver could be written. The driver is unidirectional and has no read side processing. It demonstrates some differences between module and driver programming, including the following:

Open handling	A driver is passed a minor device number or is asked to select one (see next chapter).
Flush handling	A driver must loop M_FLUSH messages back upstream.
Ioctl handling	A driver must nak ioctl messages it does not understand. This is discussed under "Driver and Module Ioctls", below.

Write side flow control is also illustrated as described above.

Driver Declarations

The driver declarations are as follows:

```
/* Simple line printer driver.  */

#include "sys/types.h"
#include "sys/param.h"
#include "sys/sysmacros.h"
#ifdef u3b2
#include "sys/psw.h"       /* required for user.h */
#include "sys/pcb.h"       /* required for user.h */
#endif
#include "sys/stream.h"
#include "sys/stropts.h"
#include "sys/dir.h"       /* required for user.h */
#include "sys/signal.h"    /* required for user.h */
#include "sys/user.h"
#include "sys/errno.h"

static struct module_info minfo = {
   0, "lp", 0, INFPSZ, 150, 50
};
```

continued

```
static int lpopen( ), lpclose( ), lpwput( );

static struct qinit rinit = {
   NULL, NULL, lpopen, lpclose, NULL, &minfo, NULL
};
static struct qinit winit = {
   lpwput, NULL, NULL, NULL, NULL, &minfo, NULL
};
struct streamtab lpinfo = { &rinit, &winit, NULL, NULL };

#define SET_OPTIONS    (('1'<<8)|1)/* really must be in a .h file */
/*
 * 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 */
};
/* Flags bits */
#define BUSY  1      * device is running and interrupt is pending */

extern struct lp lp_lp[]; /* per device lp structure array */
extern int lp_cnt;    /* number of valid minor devices */
```

As noted for modules in Chapter 6, configuring a STREAMS driver does not require the driver procedures to be externally accessible; only **streamtab** must be. All STREAMS driver procedures would typically be declared `static`.

streamtab must be defined as "*prefix*info", where *prefix* is the value of the prefix specified in the **config** file for this driver. The values in name and ID fields in the **module_info** should be unique in the system. The name field is a hook for future expansion and is not currently used. The ID is currently used only in logging and tracing (see Chapter 6 in the *Primer*). For the example in this chapter, the ID is zero.

There is no read side put or service procedure. The flow control limits for use on the write side are 50 and 150 characters. The private *lp* structure is indexed by the minor device number and contains these elements:

flags	A set of flags. Only one bit is used: BUSY indicates that output is active and a device interrupt is pending.
msg	A pointer to the current message being output.
qptr	A back pointer to the write queue. This is needed to find the write queue during interrupt processing.

Driver Open

The driver open, *lpopen,* has the same interface as the module open:

```
static int lpopen(q, dev, flag, sflag)
queue_t *q      /* read queue */
{
   struct lp *lp;

   /* Check if non-driver open */
   if (sflag)
       return OPENFAIL;

   /* Dev is major/minor */
   dev = minor(dev);
   if (dev >= lp_cnt)
       return OPENFAIL;

   /* Check if open already. q_ptr is assigned below */
   if (q->q_ptr) {
       u.u_error = EBUSY; /* only 1 user of the printer at a time */
       return OPENFAIL;
   }

   lp = &lp_lp[dev];
   lp->qptr = WR(q);
   q->q_ptr = (char *) lp;
```

continued

```
    WR(q)->q_ptr = (char *) lp;
    return dev;
}
```

The Stream flag, *sflag,* must have the value 0, indicating a normal driver open. *dev* holds both the major and minor device numbers for this port. After checking *sflag,* the open flag, *lpopen* extracts the minor device from *dev,* using the **minor**( ) macro defined in **sysmacros.h**.

NOTE The use of major devices, minor devices, and the **minor**( ) macro may be machine-dependent.

The minor device number selects a printer and must be less than *lp_cnt*.

The next check, `if (q->q_ptr)...`, determines if this printer is already open. In this case, EBUSY is returned to avoid merging print-outs from multiple users. *q_ptr* is a driver/module private data pointer. It can be used by the driver for any purpose and is initialized to zero by STREAMS. In this example, the driver sets the value of *q_ptr,* in both the read and write **queue_t** structures, to point to a private data structure for the minor device, *lp_lp[dev]*.

WR is one of three QUEUE pointer macros. As discussed in the section titled "Stream Construction" in Chapter 5, there are no physical pointers between QUEUEs, and these macros (see Appendix C) generate the pointer. **WR**(q) generates the write pointer from the read pointer, **RD**(q) generates the read pointer from the write pointer, and **OTHER**(q) generates the mate pointer from either.

Driver Processing Procedures

This example only has a write put procedure:

```
static int lpwput(q, mp)
queue_t *q;          /* write queue */
register mblk_t *mp;    /* message pointer */
{
   register struct lp *lp;
   int s;

   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);
           s = spl5( );
           /* also flush lp->msg since it is logically
            * at the head of the write queue */
           if (lp->msg) {
               freemsg(lp->msg);
               lp->msg = NULL;
           }
           splx(s);
       }

       if (*mp->b_rptr & FLUSHR) {
           flushq(RD(q), FLUSHDATA);
           *mp->b_rptr &= ~FLUSHW;
           qreply(q, mp);
       } else
           freemsg(mp);
       break;

   case M_IOCTL:
   case M_DATA:
       putq(q, mp);
       s = spl5( );
```

continued

```
        if (!(lp->flags & BUSY))
            lpout(lp);
        splx(s);
    }
}
```

Driver Flush Handling

The write put procedure, *lpwput*, illustrates driver M_FLUSH handling; note that all drivers are expected to incorporate this flush handling. If FLUSHW is set, the write message queue is flushed, and also (for this example) the leading message (`lp->msg`). **spl5** is used to protect the critical code, assuming the device interrupts at level 5. If FLUSHR is set, the read queue is flushed, the FLUSHW bit is cleared, and the message is sent upstream using **qreply**. 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 affect.

lpwput enqueues M_DATA and M_IOCTL (see the section titled "Driver and Module Ioctls" in later text) messages and, if the device is not busy, starts output by calling *lpout*. Messages types that are not recognized are discarded.

Driver Interrupt

lpintr is the driver interrupt routine:

```
/* Device interrupt routine. */

lpintr(dev)
int dev;    /* minor device number of lp */
{
   register struct lp *lp;

   lp = &lp_lp[dev];
   if (!(lp->flags & BUSY)) {
       printf("lp: unexpected interrupto);
       return;
   }
   lp->flags &= ~BUSY;
   lpout(lp);
}

/* Start output to device - used by put procedure and driver */

lpout(lp)
register struct lp *lp;
{
   register mblk_t *bp;
   queue_t *q;

   q = lp->qptr;
loop:
   if ((bp = lp->msg) == NULL) {
       if ((bp = getq(q)) == NULL)
           return;
       if (bp->b_datap->db_type == M_IOCTL) {
           lpdoioctl(lp, bp);
           goto loop;
       }
       lp->msg = bp;
   }
```

continued

```
    if (bp->b_rptr >= bp->b_wptr) {
        bp = lp->msg->b_cont;
        lp->msg->b_cont = NULL;
        freeb(lp->msg);
        lp->msg = bp;
        goto loop;
    }

    lpoutchar(lp, *bp->b_rptr++);
    lp->flags |= BUSY;
}
```

lpout simply takes a character from the queue and sends it to the printer. The processing is logically similar to the service procedure in Chapter 8. For convenience, the message currently being output is stored in `lp->msg`.

Two mythical routines need to be supplied:

lpoutchar	send a character to the printer and interrupt when complete
lpsetopt	set the printer interface options

Driver and Module Ioctls

Drivers and modules interface with **ioctl**(2) system calls through messages. Almost all STREAMS generic **ioctls** [see *streamio*(7)] go no further than the Stream head. The capability to send an **ioctl** downstream, similar to the **ioctl** of character device drivers, is provided by the I_STR **ioctl**. The Stream head processes an I_STR by constructing an M_IOCTL message (see Appendix B) from data provided in the call and sends that message downstream.

The user process that issued the I_STR is blocked until a module or driver responds with either an M_IOCACK (ack) or M_IOCNAK (nak) message, or until the request "times out" after a user-specified interval. The STREAMS module or driver that generates an ack can also return information to the process. If the Stream head does not receive one of these messages in the specified time, the **ioctl** call fails.

A module that receives an unrecognized M_IOCTL message should pass it on unchanged. A driver that receives an unrecognized M_IOCTL should nak it.

lpout traps M_IOCTL messages and calls *lpdoioctl* to process them:

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

   q = lp->qptr;

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

   switch (iocp->ioc_cmd) {
   case SET_OPTIONS:
       /* Count should be exactly one short's worth */
       if (iocp->ioc_count != sizeof(short))
           goto iocnak;
```

continued

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

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

lpdoioctl illustrates M_IOCTL processing: The first part also applies to modules. An M_IOCTL message contains a *struct iocblk* in its first block. The first block is followed by zero or more M_DATA blocks. The optional M_DATA blocks typically contain any user-supplied data.

The form of an **iocblk** is as follows:

```
struct iocblk {
    int      ioc_cmd;      /* ioctl command type */
    ushort   ioc_uid;      /* effective uid of user */
    ushort   ioc_gid;      /* effective gid of user */
    uint     ioc_id;       /* ioctl id */
    uint     ioc_count;    /* count of bytes in data field */
    int      ioc_error;    /* error code */
    int      ioc_rval;     /* return value  */
};
```

ioc_cmd contains the command supplied by the user. In this example, only one command is recognized, SET_OPTIONS. *ioc_count* contains the number of user-supplied data bytes. For this example, it must equal the size of a short (two bytes). The user data is sent directly to the printer interface using *lpsetopt*. Next, the M_IOCTL message is changed to type M_IOCACK and the *ioc_count* field is set to zero to indicate that no data is to be returned to the user. Finally, the message is sent upstream using **qreply**. If *ioc_count* was left non-zero, the Stream head would copy that many bytes from the 2nd - Nth message blocks into the user buffer.

If the M_IOCTL message is not understood or in error for any reason, the driver must set the type to M_IOCNAK and send the message upstream. No data can be sent to a user in this case. The Stream head will cause the **ioctl** call to fail with the error number EINVAL. The driver has the option of setting *ioc_error* to an alternate error number if desired.

NOTE

ioc_error can be set to a non-zero value by both M_IOCACK and M_IOCNAK. This will cause that value to be returned as an error number to the process that sent the I_STR **ioctl**.

Driver Close

The driver close clears any message being output. Any messages left on the message queue will be automatically removed by STREAMS.

```
static int lpclose(q)
queue_t *q;     /* read queue */
{
   struct lp *lp;
   int s;

   lp = (struct lp *) q->q_ptr;
   /* Free message, queue is automatically flushed by STREAMS */
   s = spl5( );
   if (lp->msg) {
       freemsg(lp->msg);
       lp->msg = NULL;
   }
   splx(s);
}
```

10 Complete Driver

Cloning 10-1

Loop-Around Driver 10-2
Write Put Procedure 10-6
Stream Head Messages 10-10
Service Procedures 10-10
Close 10-13

Cloning

The clone mechanism has been developed as a convenience. It allows a user to open a driver without specifying the minor device. When a Stream is opened, a flag indicating a clone open is tested by the driver open routine. If the flag is set, the driver returns an unused minor device number. The clone driver [see *clone*(7)] is a system-dependent STREAMS pseudo-driver.

Knowledge of clone driver implementation is not required to use it. A description is presented here for completeness and to assist developers who must implement their own clone driver. A cloneable device has a device number in which the major number corresponds to the clone driver and the minor number corresponds to the target driver. When an **open**(2) system call is made to the associated (STREAMS) file, **open** causes a new Stream to be opened to the **clone** driver and the open procedure in **clone** to be called with *dev* set to *clone/target*. The **clone** open procedure uses `minor(dev)` to locate the **cdevsw** entry of the target driver. Then, **clone** modifies the contents of the newly created Stream **queue_ts** to those of the target driver and calls the target driver open procedure with the Stream flag set to CLONEOPEN. The target driver open responds to the CLONEOPEN by returning an unused minor device number. When the **clone** open receives the returned target driver minor device number, it allocates a new inode (which has no name in the file system) and associates the minor device number with the inode.

Loop-Around Driver

The loop-around driver is a pseudo-driver that loops data from one open Stream to another open Stream. The user processes see the associated files as a full duplex pipe. The Streams are not physically linked. The driver is a simple multiplexer (see next chapter), which passes messages from one Stream's write QUEUE to the other Stream's read QUEUE.

To create a pipe, 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 L_STR **ioctl** to the other Stream. For each **open**, the driver open places the passed **queue_t** pointer in a driver interconnection table, indexed by the device number. When the driver later receives the L_STR as an M_IOCTL message, it uses the device number to locate the other Stream's interconnection table entry and stores the appropriate **queue_t** 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 10-1. Flow control between the two Streams must be handled by special code since STREAMS will not automatically propagate flow control information between two Streams that are not physically interconnected.

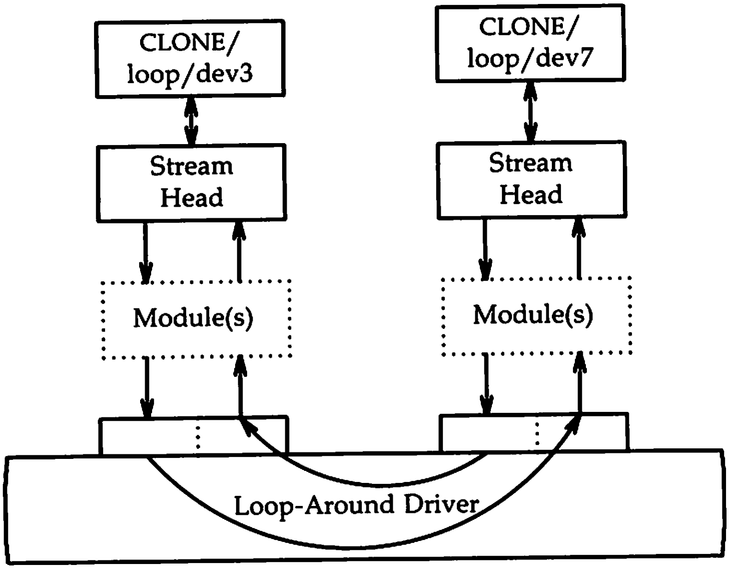
CLONE/
loop/dev3
CLONE/
loop/dev7
Stream
Head
Stream
Head
Module(s)
Module(s)
Loop-Around Driver

Figure 10-1: Loop-Around Streams

The declarations for the driver are:

```
/*
 * Loop-around driver
 */

#include "sys/types.h"
#include "sys/param.h"
#include "sys/sysmacros.h"
#ifdef u3b2
#include "sys/psw.h"
#include "sys/pcb.h"
#endif
#include "sys/stream.h"
```

continued

```
#include "sys/stropts.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"

static struct module_info minfo = {
   0, "loop", 0, INFPSZ, 512, 128
};

static int loopopen( ), loopclose( ), loopwput( ), loopwsrv( ), looprsrv( );

static struct qinit rinit = {
   NULL, looprsrv, loopopen, loopclose, NULL, &minfo, NULL
};

static struct qinit winit = {
   loopwput, loopwsrv, NULL, NULL, NULL, &minfo, NULL
};

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_SET    (('1'<<8)|1)    /* should be in a .h file */

extern struct loop loop_loop[ ];
extern int loop_cnt;
```

A **config** file to configure the *loop* driver is shown in Appendix E. 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 address of the corresponding *loop_loop* element is placed in *q_ptr* (private data structure pointer) of the read and write side **queue_ts**. Since STREAMS clears *q_ptr* when the **queue_t** is allocated, a NULL value of *q_ptr* indicates an initial **open**. *loop_loop* is used to verify that this Stream is connected to another open Stream.

The open procedure includes canonical clone processing which enables a single file system node to yield a new minor device/inode each time the driver is opened:

```
static int loopopen(q, dev, flag, sflag)
queue_t *q;
{
    struct loop *loop;

    /*
     * If CLONEOPEN, pick a minor device number to use.
     * Otherwise, check the minor device range.
     */
    if (sflag == CLONEOPEN) {
        for (dev = 0; dev < loop_cnt; dev++) {
            if (loop_loop[dev].qptr == NULL)
                break;
        }
    }
    else
        dev = minor(dev);

    if (dev >= loop_cnt)
        return OPENFAIL;  /* default = ENXIO */

    /* Setup data structures */

    if (q->q_ptr)  /* already open */
        return dev;

    loop = &loop_loop[dev];
    WR(q)->q_ptr = (char *) loop;
    q->q_ptr = (char *) loop;
    loop->qptr = WR(q);

    /*
     * The return value is the minor device.
     * For CLONEOPEN case, this will be used for
     * newly allocated inode
     */
    return dev;
}
```

In *loopopen, sflag* can be CLONEOPEN, indicating that the driver should pick a minor device (i.e., the user does not care which minor device is used). In this case, the driver scans its private *loop_loop* data structure to find an unused minor device number. If *sflag* has not been set to CLONEOPEN, the passed-in minor device is used.

The return value is the minor device number. In the CLONEOPEN case, this value will be used by the **clone** driver for the newly allocated inode and will then be passed to the user.

Write Put Procedure

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:

```
static int loopwput(q, mp)
queue_t *q;
mblk_t *mp;
{
   register struct loop *loop;

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

   switch (mp->b_datap->db_type) {
   case M_IOCTL: {
       struct iocblk *iocp;
       int error;

       iocp = (struct iocblk *)mp->b_rptr;
       switch (iocp->ioc_cmd) {
       case LOOP_SET: {
           int to; /* other minor device */
           /*
            * 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;
           }
```

continued

```
        /* fetch other dev from 2nd message block */

        to = *(int *)mp->b_cont->b_rptr;

        /*
         * More sanity checks.  The minor must be in range, open already.
         * Also, this device and the other one must be disconnected.
         */

        if (to >= loop_cnt || to < 0 || !loop_loop[to].qptr) {
            error = ENXIO;
            goto iocnak;
        }

        if (loop->oqptr || loop_loop[to].oqptr) {
            error = EBUSY;
            goto iocnak;
        }

        /*
         * Cross-connect streams via the loop structures
         */

        loop->oqptr = RD(loop_loop[to].qptr);
        loop_loop[to].oqptr = RD(q);

        /*
         * Return successful ioctl.  Set ioc_count
         * to zero, since there is return no data.
         */

        mp->b_datap->db_type = M_IOCACK;
        iocp->ioc_count = 0;
        qreply(q, mp);
        break;
    }

    default:
        error = EINVAL;
    iocnak:
```

continued

```
            /*
             * Bad ioctl.  Setting ioc_error causes the
             * ioctl call to return that particular errno.
             * By default, ioctl will return EINVAL on failure
             */
            mp->b_datap->db_type = M_IOCNAK;
            iocp->ioc_error = error;  /* set returned errno */
            qreply(q, mp);

        }
        break;

    }
```

loopwput shows another use of an I_STR **ioctl** call (see the section titled "Driver and Module Ioctls" in Chapter 9). The driver supports a LOOP_SET value of *ioc_cmd* in the *iocblk* of the M_IOCTL message. LOOP_SET instructs the driver to connect the current open Stream to the Stream indicated in the message. The second block of the 1M_IOCTL message holds an integer that specifies the minor device number of the Stream to connect to.

The driver performs 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 everything checks out, the read **queue_t** pointers for the two Streams are stored in the respective *oqptr* fields. This cross-connects the two Streams indirectly, via *loop_loop*.

Canonical flush handling is incorporated in the put procedure:

```
    case M_FLUSH:
        if (*mp->b_rptr & FLUSHW)
            flushq(q, 0);
        if (*mp->b_rptr & FLUSHR) {
            flushq(RD(q), 0);
            *mp->b_rptr &= ~FLUSHW;
            qreply(q, mp);
        } else
            freemsg(mp);
        break;
    default:
        /*
         * If this stream isn't connected, send an M_ERROR upstream.
         */
        if (loop->oqptr == NULL) {
            putctl1(RD(q)->q_next, M_ERROR, ENXIO);
            freemsg(mp);
            break;
        }
        putq(q, mp);
    }
}
```

Finally, *loopwput* enqueues all other messages (e.g., M_DATA or M_PROTO) for processing by its service procedure. A check is made to see if the Stream is connected. If not, an M_ERROR is sent upstream to the Stream head (see below).

putctl1 and **putctl** (see below) are utilities that allocate a non-data (i.e., not M_DATA, M_PROTO, or M_PCPROTO) type message; place one byte in the message (for **putctl1**) and call the put procedure of the specified QUEUE (see Appendix C).

Stream Head Messages

Certain message types (see Appendix B) can be sent upstream by drivers and modules to the Stream head where they are translated into actions detectable by user process(es). 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** and **poll** will fail. Also causes an 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 will fail. Once the Stream head read message queue is empty, EOF is returned on reads. Can also result in SIGHUP signal to the process group.
M_SIG/M_PCSIG	Causes a specified signal to be sent to a process (see Chapter 13).

Service Procedures

Service procedures are required on both the write and read sides for purposes of flow control:

```
static int loopwsrv(q)
register queue_t *q;
{
    mblk_t *mp;
    register struct loop *loop;

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

    while ((mp = getq(q)) != NULL) {

        /*
         * Check if we can put the message up the other stream read queue
         */

        if (mp->b_datap->db_type <= QPCTL && !canput(loop->oqptr->q_next)) {
            putbq(q, mp);  /* read side is blocked */
            break;
        }

        /* send message */

        putnext(loop->oqptr, mp);  /* To queue following other stream read queue */
    }
}

static int looprsrv(q)
queue_t *q;

{

/*  Enter only when "back-enabled" by flow control */

    struct loop *loop;

    loop = (struct loop *)q->q_ptr;
    if (loop->oqptr == NULL)
        return;

    /* manually enable write service procedure */

    qenable(WR(loop->oqptr));
}
```

The write service procedure, *loopwsrv*, takes on the canonical form (see Chapter 8) with a difference. The QUEUE being written to is not downstream, but upstream (found via *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 becomes unstuck from flow control blockage. The purpose of the procedure is to re-enable the writer (*loopwsrv*) by using *oqptr* to find the related **queue_t**. *loopwsrv* cannot be directly back-enabled by STREAMS because there is no direct **queue_t** linkage between the two Streams. Note that no message ever gets queued to the read service procedure. Messages are kept on the write side so that flow control can propagate up to the Stream head. There is a defensive check to see if the cross-connect has broken. **qenable** schedules the write side of the other Stream.

Close

loopclose breaks the connection between the Streams.

```
static int loopclose(q)
queue_t *q;
{
   register 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 fail writes,
    * allow the queued data to be read completely, and then
    * return EOF on subsequent reads.
    */

   if (loop->oqptr) {
       ((struct loop *)loop->oqptr->q_ptr)->qptr = NULL;
       ((struct loop *)loop->oqptr->q_ptr)->oqptr = NULL;
       putctl(loop->oqptr->q_next, M_HANGUP);
       loop->oqptr = NULL;
   }
}
```

loopclose sends an M_HANGUP message (see above) up the connected Stream to the Stream head.

NOTE

This driver can be implemented much more cleanly by actually linking the *q_next* pointers of the **queue_t** pairs of the two Streams.

11 Multiplexing

Multiplexing Configurations 11-1
Connecting Lower Streams 11-2
Disconnecting Lower Streams 11-4

Multiplexer Construction Example 11-5

Multiplexing Driver 11-8
Upper Write Put Procedure 11-12
Lower QUEUE Write Service Procedure 11-16
Lower Read Put Procedure 11-19

Multiplexing Configurations

This chapter describes how STREAMS multiplexing configurations are created and discusses multiplexing drivers. A STREAMS multiplexer is a pseudo-driver with multiple Streams connected to it. The primary function of the driver is to switch messages among the connected Streams. Multiplexer configurations are created from user level by system calls. Chapter 6 of the *Primer* contains the required introduction to STREAMS multiplexing.

STREAMS-related system calls are used to set up the "plumbing," or Stream interconnections, for multiplexing pseudo-drivers. The subset of these calls that allows a user to connect (and disconnect) Streams below a pseudo-driver is referred to as the multiplexing facility. This type of connection will be referred to as a 1-to-M, or lower, multiplexer configuration (see Figure 6-2 in the *Primer*). This configuration must always contain a multiplexing pseudo-driver, which is recognized by STREAMS as having special characteristics.

Multiple Streams can be connected above a driver by use of **open** calls. This was done for the loop-around driver of the previous chapter 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 will be referred to as an N-to-1, or upper, multiplexer (see Figure 6-1 in the *Primer*). STREAMS does not provide any facilities beyond **open** and **close** to connect or disconnect upper Streams for multiplexing purposes.

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 multiple Streams.

M-to-N multiplexing configurations are implemented by using both of the above mechanisms in a driver. Complex multiplexing trees can be created by cascading multiplexing Streams below one another.

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 are not really connected to their common driver. 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. As stated above, the only difference between configurations is the manner of connecting and disconnecting. Only lower connections have use of the multiplexing facility.

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. As usual, **open** uses the first two **streamtab** structure entries (see the section titled "Opening a Stream," in Chapter 5) to create the driver QUEUEs. At this point, the only distinguishing characteristic of this Stream are non-NULL entries in the **streamtab** *st_mux[rw]init* (mux) fields:

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

These fields are ignored by the **open** (see the rightmost Stream in Figure 11-1). 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 (see the leftmost Stream in Figure 11-1). 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** call [see *streamio*(7)]. As shown in Figure 5-1, all Stream components are constructed in a similar manner. The Stream head points to the stream-head-routines as its procedures (known via its **queue_t**). An I_LINK to the upper Stream, referencing the lower Stream, causes STREAMS to modify the contents of the Stream head in the lower Stream. The pointers to the stream-head-routines, and other values, in the Stream head 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 will, ultimately, be passed to the put procedure designated in *st_muxrinit*, the multiplexing driver. The L_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 to the multiplexing driver an M_IOCTL message with *ioc_cmd* set to I_LINK (see discussions of the **iocblk** structure in Chapter 9 and Appendix A). The M_DATA part of the M_IOCTL contains a **linkblk** structure:

```
struct linkblk {
   queue_t  *l_qtop;   /* lowest level write queue of upper stream */
   queue_t  *l_qbot;   /* highest level write queue of lower stream */
   int       l_index;  /* system-unique index for lower stream. */
};
```

The multiplexing driver stores information from the **linkblk** in private storage and returns an M_IOCACK message (ack). *l_index* is returned to the process requesting the I_LINK. This value can be used later by the process to disconnect this Stream, as described below. **linkblk** contents are further discussed below.

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 process(es) along any of the upper Streams. The upper Stream(s) provides the only interface between the user process(es) and the multiplexer.

Note that no direct data structure linkage is established for the linked Streams. The *q_next* pointers of the lower Stream still appear to connect with a Stream head. Messages flowing upstream from a lower driver (a device driver or another multiplexer) will enter the multiplexing driver (i.e., Stream head) put procedure with *l_qbot* as the **queue_t** 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 the control, or any other, upper Stream has to be processed and routed, if required, by the driver.

Also note that the lower Stream (see the headers and file descriptors in Figure 11-2) is no longer accessible from user space. This causes all system calls to the lower Stream to return EINVAL, with the exception of **close**. This is why all modules have to be in place before the lower Stream is linked to the multiplexing driver. As a general rule, the lower Stream file should be closed after it is linked (see following section). This does not disturb the multiplexing configuration.

Finally, note that the absence of direct linkage between the upper and lower Streams means that STREAMS flow control has to be handled by special code in the multiplexing driver. The flow control mechanism cannot see across 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 developer-dependent. However, there is a system limit, NMUXLINK (see Appendix E), to the number of Streams that can be linked in the system.

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** referencing a specific Stream, an I_UNLINK indicating all lower Streams, or the last **close** (i.e., causes the associated file to be closed) of the control Stream. As in the link, an unlink sends a **linkblk** 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 the file descriptor for a lower Stream was previously closed, a subsequent unlink will automatically close the Stream. Otherwise, the lower Stream must be closed by **close** following the unlink. STREAMS will automatically dismantle all cascaded multiplexers (below other multiplexing Streams) if their controlling Stream is closed. An I_UNLINK will leave 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. A multiplexing configuration similar to the Internet of Figure 6-2 in the *Primer* is discussed. Figure 11-1 shows the Streams before their connection to create the multiplexing configuration of Figure 11-2. Multiple upper and lower Streams interface to the multiplexer driver. The user processes of Figure 11-2 are not shown in Figure 11-1.

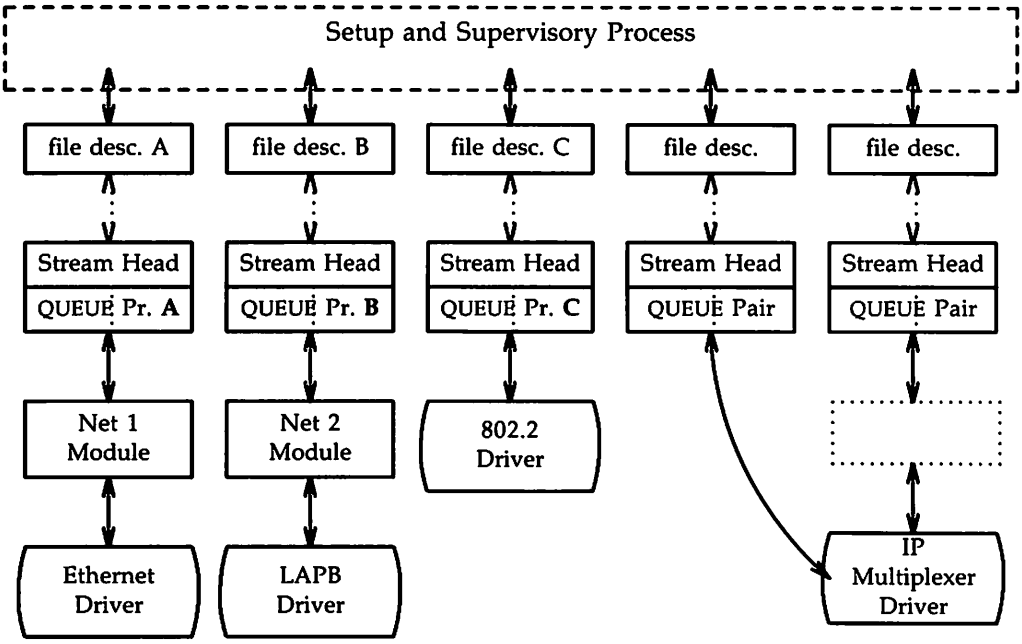
Setup and Supervisory Process
file desc. A
file desc. B
file desc. C
file desc.
file desc.
Stream Head
QUEUE Pr. A
Stream Head
QUEUE Pr. B
Stream Head
QUEUE Pr. C
Stream Head
QUEUE Pair
Stream Head
QUEUE Pair
Net 1 Module
Net 2 Module
802.2 Driver
Ethernet Driver
LAPB Driver
IP Multiplexer Driver

Figure 11-1: Internet Multiplexer Before Connecting

The Ethernet, LAPB, and IEEE 802.2 device drivers terminate links to other nodes. IP (Internet Protocol) is a multiplexer driver. IP switches datagrams among the various nodes or sends them upstream to a user(s) in the system. The Net modules would typically provide a convergence function which matches the IP and device driver interface.

Figure 11-1 depicts only a portion of the full, larger Stream. As shown in the dotted rectangle above the IP multiplexer, there generally would be an upper TCP multiplexer, additional modules and, possibly, additional multiplexers in the Stream. Multiplexers could also be cascaded below the IP driver if the device drivers were replaced by multiplexer drivers.

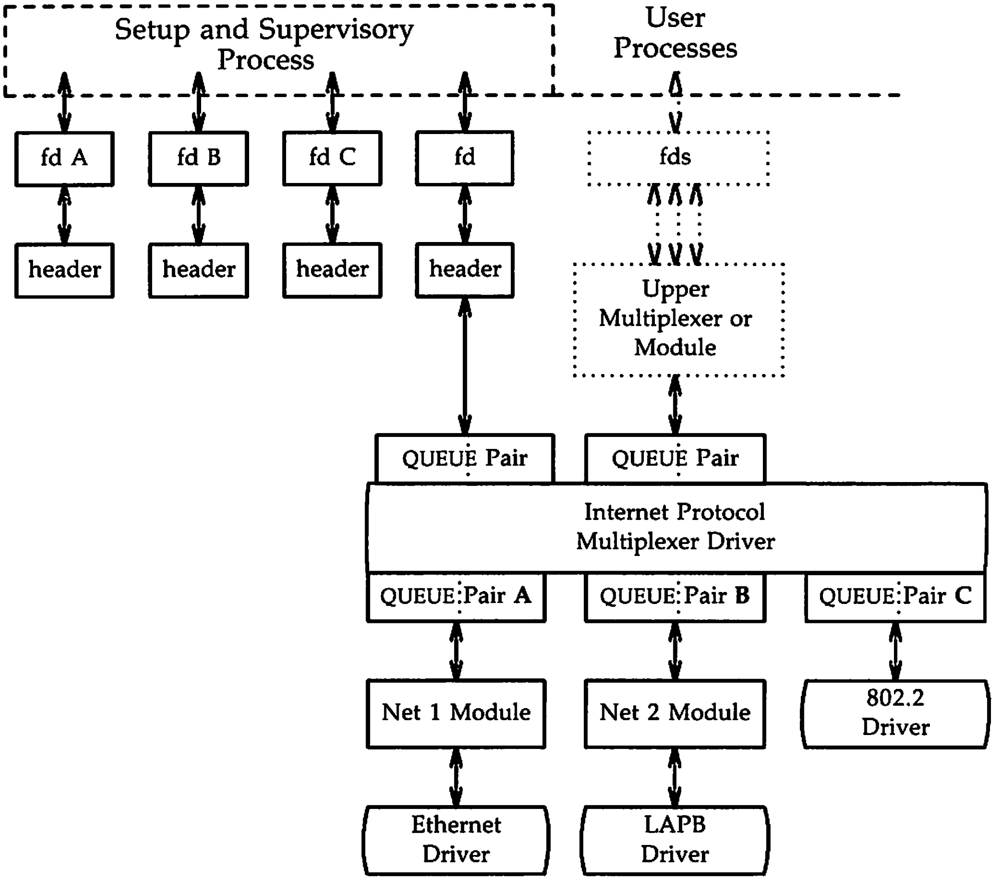
Setup and Supervisory Process
User Processes
fd A
fd B
fd C
fd
fds
header
header
header
header
Upper Multiplexer or Module
QUEUE Pair
QUEUE Pair
Internet Protocol Multiplexer Driver
QUEUE:Pair A
QUEUE:Pair B
QUEUE:Pair C
Net 1 Module
Net 2 Module
802.2 Driver
Ethernet Driver
LAPB Driver

Figure 11-2: Internet Multiplexer After Connecting

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 which 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.

As shown in Figure 11-2, the file descriptors for the lower device driver Streams are left dangling. The primary purpose in creating these Streams was to provide parts for the multiplexer. Those not used for control and not required for error recovery (by reconnecting them through an I_UNLINK **ioctl**) have no further function. As stated above, these lower Streams can be closed to free the file descriptor without any effect on the multiplexer. A setup process installing a configuration containing a large number of drivers should do this to avoid running out of file descriptors.

Multiplexing Driver

This section contains an example of a multiplexing driver that implements an N-to-1 configuration, similar to that of Figure 6-3 in the *Primer*. 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 (i.e., relevant user process) is contained in the input data rather than in an interrupt call.

Each upper Stream is connected by an **open**, identical to the driver of Chapter 9. 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, 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.

The multiplexer declarations are:

```
#include "sys/types.h"
#include "sys/param.h"
#include "sys/sysmacros.h"
#include "sys/stream.h"
#include "sys/stropts.h"
#include "sys/errno.h"

static int muxopen( ), muxclose( ), muxuwput( ), muxlwsrv( ), muxlrput( );

static struct module_info info = {
   0, "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, NULL, 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 */
};

extern struct mux mux_mux[ ];
extern int mux_cnt;

queue_t *muxbot;    /* linked lower queue */
int muxerr;         /* set if error of hangup on lower stream */

static queue_t *get_next_q( );
```

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 completed 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 a lower Stream read put procedure. In general, only upper write side and lower read side procedures are used. Application specific flow control requirements might modify this. If the QUEUE procedures of the opposite upper/lower QUEUE are not needed, the QUEUE can be skipped over 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 service procedure, as described below.

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 minor device for a CLONEOPEN driver open case.

The upper QUEUE open contains the canonical driver open code:

```
static int muxopen(q, dev, flag, sflag)
queue_t *q;
{
    struct mux *mux;

    if (sflag == CLONEOPEN) {
        for (dev = 0; dev < mux_cnt; dev++)
            if (mux_mux[dev].qptr == 0)
                break;
        }
    }
    else
        dev = minor(dev);

    if (dev >= mux_cnt)
        return OPENFAIL;

    mux = &mux_mux[dev];
    mux->qptr = q;
    q->q_ptr = (char *) mux;
    WR(q)->q_ptr = (char *) mux;
    return dev;
}
```

muxopen checks for a clone or ordinary open call. It loads *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. This allows flow control to propagate towards the Stream head for each upper Stream. However, there is no service procedure on the upper write side. All M_DATA messages from all the upper message queues are ultimately dequeued by the service procedure on the lower (linked) write side. The upper write Streams are serviced in a round-robin fashion by the lower write service procedure. 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 is 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. That is, the put procedure treats the Stream component above the driver as the next QUEUE.

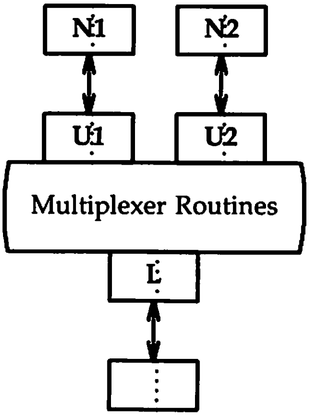

Figure 11-3: Example Multiplexer Configuration

This is shown in Figure 11-3. Multiplexer Routines are all the above procedures. U1 and U2 are **queue_t** pairs, each including a write **queue_t** pointed at by an *l_qtop* in a **linkblk** (see beginning of this chapter). L is the **queue_t** pair which contains the write **queue_t** pointed at by *l_qbot*. N1 and N2 are the modules (or Stream head or another multiplexing driver) seen by L when read side messages are sent upstream.

Upper Write Put Procedure

The upper QUEUE write put procedure, *muxuwput*, traps ioctls, in particular I_LINK and I_UNLINK:

```
static int muxuwput(q, mp)
queue_t *q;
mblk_t *mp;

{

    int s;
    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:
```

continued

```
        /*
         * Unlink.  The data contains a linkblk structure.
         * 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;

}
```

First, there is a check to enforce that the Stream associated with minor device 0 will be the single, controlling Stream. Ioctls 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, described previously, containing:

l_qtop The upper write QUEUE from which the ioctl is coming. It should always equal *q*.

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, as described above. Since the example only supports a single link, *l_index* is not used.

For L_LINK, *l_qbot* is saved in *muxbot* and an ack is generated. From this point on, until an L_UNLINK occurs, data from upper queues will be routed through *muxbot*. Note that when an L_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 (nak) in response to an L_LINK will cause the lower Stream to be disconnected.

The L_UNLINK handling code nulls out *muxbot* and generates an ack. A nak should not be returned to an L_UNLINK. The Stream head assures that the lower Stream is connected to a multiplexer before sending an L_UNLINK M_IOCTL.

muxuwput handles M_FLUSH messages as a normal driver would:

```
case M_FLUSH:
    if (*mp->b_rptr & FLUSHW)
        flushq(q, FLUSHDATA);
    if (*mp->b_rptr & FLUSHR) {
        flushq(RD(q), FLUSHDATA);
        *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.
     */
    if (muxerr || muxbot == NULL)
        goto bad;
```

continued

```
        putq(q, mp);   /* place message on upper write message queue */
        qenable(muxbot);   /* lower service write procedure */
        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);
    }
}
```

M_DATA messages are not placed on the lower write message queue. They are queued on the upper write message queue. **putq** recognizes the absence of the upper service procedure and does not schedule the QUEUE. Then, the lower service procedure, *muxlwsrv* is scheduled with **qenable** (see Appendix C) to start output. This is similar to starting output on a device driver. Note that *muxuwput* cannot access *muxlwsrv* (the lower QUEUE write service procedure, contained in *muxbot*) by the conventional STREAMS calls, **putq** or **putnext** (to a *muxlwput*). Both calls require that a message be passed, but the messages remain on the upper Stream.

Lower QUEUE Write Service Procedure

The lower (linked) queue write service procedure *muxlwsrv,* is scheduled directly from the upper service procedures. It is also scheduled from the lower Stream, by being back-enabled when the lower Stream becomes unblocked from downstream flow control.

```
static int muxlwsrv(q)
register queue_t *q;
{
   register mblk_t *mp, *bp;
   register queue_t *nq;

   /*
    * While lower stream is not blocked, find an upper queue to
    * service (get_next_q) and send one message from it downstream.
    */
   while (canput(q->q_next)) {
       nq = get_next_q( );
       if (nq == NULL)
           break;
       [illegible] = getq(nq);
       /*
        * Prepend the outgoing message with a single byte header
        * that indicates the minor device number it came from.
        */
       if ((bp = allocb(1, BPRI_MED)) == NULL) {
           printf("mux: allocb failed (size 1)\n");
           freemsg(mp);
           continue;
       }
       *bp->b_wptr++ = (struct mux *)nq->q_ptr - mux_mux;
       bp->b_cont = mp;
       putnext(q, bp);
   }
}
```

muxlwsrv takes data from the upper queues and puts it out through *muxbot*. The algorithm used is simple round robin. While we can put to `muxbot->q_next`, we select an upper QUEUE (via *get_next_q)* and move a message from it to *muxbot*. Each message is prefixed by a one-byte header that indicates which upper Stream it came from.

Finding messages on upper write queues is handled by *get_next_q*:

```
/*
 * Round-robin scheduling.
 * Return next upper queue that needs servicing.
 * Returns NULL when no more work needs to be done.
 */

static queue_t *
get_next_q( )
{
    static int next;
    int i, start;
    register queue_t *q;

    start = next;
    for (i = next; i < mux_cnt; i++)
        if (q = mux_mux[i].qptr) {
            q = WR(q);
            if (q->q_first) {
                next = i+1;
                return q;
            }
        }

    for (i = 0; i < start; i++)
        if (q = mux_mux[i].qptr) {
            q = WR(q);
            if (q->q_first) {
                next = i+1;
                return q;
            }
        }

    return NULL;
}
```

get_next_q searches the upper queues in a round-robin fashion looking for the first one containing a message. It returns the **queue_t** pointer or NULL if there is no work to do.

Lower Read Put Procedure

The lower (linked) queue read put procedure is:

```
static int muxlrput(q, mp)
queue_t *q;
mblk_t *mp;
{
   queue_t *uq;
   mblk_t *b_cont;
   int dev;

   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 & FLUSHR)
           flushq(q, 0);
       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:
```

continued

```
        /*
         * 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.
         * The stream head interprets a leading zero-length block
         * as an EOF regardless of what follows (sigh).
         */

        dev = *mp->b_rptr++;
        if (mp->b_rptr == mp->b_wptr && (b_cont = mp->b_cont)) {
            freeb(mp);
            mp = b_cont;
        }

        /* Sanity check.  Device must be in range */

        if (dev < 0 || dev >= mux_cnt) {
            freemsg(mp);
            break;
        }

        /*
         * If upper stream is open and not backed up,
         * send the message there, otherwise discard it.
         */

        uq = mux_mux[dev].qptr;
        if (uq != NULL && canput(uq->q_next))
            putnext(uq, mp);
        else
            freemsg(mp);
        break;
    default:
        freemsg(mp);
    }
}
```

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 reversed from that of a driver, handling M_FLUSH messages from upstream.

muxlrput also handles M_ERROR and M_HANGUP messages. If one is received, it locks up the upper Streams.

M_DATA messages are routed by looking at the first data byte of the message. This byte contains the minor device of the upper Stream. If removing this byte causes the leading block to be empty and more blocks follow, the block is discarded. This is done because the Stream head interprets a leading zero-length block as an EOF [see *read*(2)]. Several sanity checks are made: Does the message have at least one byte? Is the device in range? Is the upper Stream open? Is the upper Stream not full?

This mux does not do end-to-end flow control. It is merely a router (like the Department of Defense's IP protocol). If everything checks out, the message is put to the proper upper QUEUE. Otherwise, the message is silently discarded.

The upper Stream close routine simply clears the mux entry so this queue will no longer be found by *get_next_queue*:

```
/*
 * Upper queue close
 */
static int muxclose(q)
queue_t *q;
{
   ((struct mux *)q->q_ptr)->qptr = NULL;
}
```

12 Service Interface

Definition 12-1
Message Usage 12-1

Example 12-3
Declarations 12-3
Service Interface Procedure 12-5

Definition

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 4-2). A service interface is a set of primitives and the rules for the allowable sequences of primitives across the boundary. 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 (generically, PROTO) messages. M_PCPROTO is the priority version of M_PROTO.

Message Usage

As described in Appendix B, PROTO messages 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 and 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** will compact 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 may be processed in a module that receives it, or it may 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 (up and downstream) on a Stream and bidirectionally between a Stream and a user process. **putmsg** and **getmsg** system calls are analogous, respectively, to **write** and **read** except that the former allow both data and control parts to be (separately) passed, and they observe message boundary alignment across the user-Stream boundary. **putmsg** and **getmsg** 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 M_PROTO messages and not to carry protocol expedited data. M_PCPROTO insures 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**.

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 may be inserted between a service user and a service provider to manipulate the data part as it passes between them. However, these modules may not 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 may be changed, but the message may not be split into separate messages nor combined with other messages.

In addition, modules and drivers must observe the rule that priority messages are not subject to flow control and forward them accordingly (see the beginning of *modwsrv* in Chapter 8). Priority messages also bypass flow control at the user-Stream boundary [see *putmsg*(2)].

Example

The example below is part of a module which illustrates the concept of a service interface. The module implements a simple datagram interface and mirrors the example in Chapter 4.

Declarations

The service interface primitives are defined in the declarations:

```
#include "sys/types.h"
#include "sys/param.h"
#include "sys/stream.h"
#include "sys/errno.h"

/*
 * Primitives initiated by the service user:
 */
#define BIND_REQ      1  /* bind request  */
#define UNITDATA_REQ 2  /* unitdata request   */
/*
 * Primitives initiated by the service provider:
 */
#define OK_ACK        3  /* bind 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 */
};
```

continued

```
struct error_ack {    /* error acknowledgment */
   long  PRIM_type;   /* always ERROR_ACK */
   long  UNIX_error;  /* UNIX 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 per 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.

Two commands are recognized by the module:

BIND_REQ	Give this Stream a protocol address, that is, give it a name on the network. After a BIND_REQ is completed, datagrams from other senders will find their way through the network to this particular Stream.

UNITDATA_REQ	Send a datagram to the specified address.

Three messages are generated:

OK_ACK	A positive acknowledgment (ack) of BIND_REQ.
ERROR_ACK	A negative acknowledgment of BIND_REQ.
UNITDATA_IND	A datagram from the network has been received. (This code is not shown.)

The ack 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 insure 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 user would get confused.

The driver uses a per-minor device data structure, *dgproto,* which contains the following:

state	current state of the Stream (endpoint) 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 one of these structures.

Service Interface Procedure

The write put procedure is:

```
static int protowput(q, mp)
queue_t *q;
mblk_t *mp;
{
   union primitives *proto;
   struct dgproto *dgproto;
   int err;

   dgproto = (struct dgproto *) q->q_ptr;
```

continued

```
switch (mp->b_datap->db_type) {
default:
    /* don't understand it */
    mp->b_datap->db_type = M_ERROR;
    mp->b_rptr = mp->b_wptr = mp->b_datap->db_base;
    *mp->b_wptr++ = EPROTO;
    qreply(q, mp);
    break;
case M_FLUSH:
    /* standard flush handling goes here ... */
    break;
case M_PROTO:
    /* Protocol message -> user request */

    proto = (union primitives *) mp->b_rptr;

    switch (proto->type) {
    default:
        mp->b_datap->db_type = M_ERROR;
        mp->b_rptr = mp->b_wptr = mp->b_datap->db_base;
        *mp->b_wptr++ = EPROTO;
        qreply(q, mp);
        return;

    case BIND_REQ:
        if (dgproto->state != IDLE) {
            err = EINVAL;
            goto error_ack;
        }
        if (mp->b_wptr - mp->b_rptr != sizeof(struct bind_req)) {
            err = EINVAL;
            goto error_ack;
        }
        if (err = chkaddr(proto->bind_req.BIND_addr))
            goto error_ack;

        dgproto->state = BOUND;
        dgproto->addr = proto->bind_req.BIND_addr;
        mp->b_datap->db_type = M_PCPROTO;
        proto->type = OK_ACK;
        mp->b_wptr = mp->b_rptr + sizeof(struct ok_ack);
        qreply(q, mp);
        break;
```

continued

```
        error_ack:
            mp->b_datap->db_type = M_PCPROTO;
            proto->type = ERROR_ACK;
            proto->error_ack.UNIX_error = err;
            mp->b_wptr = mp->b_rptr + sizeof(struct error_ack);
            qreply(q, mp);
            break;

        case UNITDATA_REQ:
            if (dgproto->state != BOUND)
                goto bad;
            if (mp->b_wptr - mp->b_rptr != sizeof(struct unitdata_req))
                goto bad;
            if (err = chkaddr(proto->unitdata_req.DEST_addr))
                goto bad;
            if (mp->b_cont) {
                putq(q, mp->b_cont);

                /* start device or mux output ... */
            }

            break;
        bad:
            freemsg(mp);
            break;
        }
    }
}
```

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 will perform 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. (This action may seem rash, but it is in accordance with the interface specification, which is not shown. Another specification might call for the generation of a UNITDATA_ERROR indication.) If all is well, the data part of the message, if it exists, is put on 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.

Another piece of code not shown is the generation of UNITDATA_IND messages. This would normally occur in the device interrupt if this is a hardware driver (like STARLAN) or in the lower read put procedure if this is a multiplexer. The algorithm is simple: The data part of the message is prepended by an M_PROTO message block that contains a *unitdata_ind* structure and sent upstream.

13 Advanced Topics

Recovering From No Buffers 13-1

Advanced Flow Control 13-5

Signals 13-6

Control of Stream Head Processing 13-8

Read Options 13-8

Write Offset 13-9

Recovering From No Buffers

The **bufcall** utility (see Appendix C) is used to recover from an **allocb** failure. The call syntax is as follows:

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

bufcall will call (**func*)(*arg*) when a buffer of *size* bytes at *pri* priority is available. When *func* is called, it has no user context and must return without sleeping. Also, because of interrupt processing, there is no guarantee that when *func* is called, a buffer will actually be available (someone else may steal it). **bufcall** returns 1 on success, indicating that the request has been successfully recorded, or 0 on failure. On a failure return, the requested function will never be called.

CAUTION

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

Two examples are provided. Example one is a device receive interrupt handler:

```
#include "sys/types.h"
#include "sys/param.h"
#include "sys/stream.h"

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;

    if ((bp = allocb(DEVBLKSZ, BPRI_MED)) == NULL) {
        printf("dev: allocb failure (size %d)\n", DEVBLKSZ);
        /*
         * Allocation failed.  Use bufcall to
         * schedule a call to ourself.
         */
        (void) bufcall(DEVBLKSZ, BPRI_MED, dev_re_load, dev);
        return;
    }

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

dev_rintr is called when the device has posted a receive interrupt. The code retrieves the data from the device (not shown). *dev_rintr* must then give the device another buffer to fill by a call to *dev_re_load,* which calls **allocb** with the appropriate buffer size (DEVBLKSZ, definition not shown) and priority. If **allocb** fails, *dev_re_load* uses **bufcall** to call itself when STREAMS determines a buffer of the appropriate size and priority is available.

NOTE

Since **bufcall** may fail, there is still a chance that the device may hang. A better strategy, in the event **bufcall** fails, would be to discard the current input message and resubmit that buffer to the device. Losing input data is generally better than hanging.

The second example is a write service procedure, *mod_wsrv,* which needs to prepend each output message with a header (similar to the multiplexer example of Chapter 11). *mod_wsrv* illustrates a case for potential deadlock:

```
static int mod_wsrv(q)
queue_t *q;
{
   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 ourself.
        */
       if ((bp = allocb(HDRSZ, BPRI_MED)) == NULL) {
           if (!bufcall(HDRSZ, BPRI_MED, qenable, q)) {
               /*
                * The bufcall request has failed.  Discard
                * the message and keep running to avoid hanging.
                */
               freemsg(mp);
               continue;
           }
           /*
            * Put the message back and exit, we will be re-enabled later
            */
           putbq(q, mp);
           return;
       }

       /* process message .... */

   }
}
```

However, if **allocb** fails, *mod_wsrv* wants to recover without loss of data ands calls **bufcall**. In this case, the routine passed to **bufcall** is **qenable** (see below and Appendix C). When a buffer is available (of size HDRSZ, definition not shown), the service procedure will be automatically re-enabled. Before exiting, the current message is put back on the queue. This example deals with **bufcall** failure by discarding the current message and continuing in the service procedure loop.

Advanced Flow Control

Streams provides mechanisms to alter the normal queue scheduling process. **putq** will not schedule a QUEUE if **noenable**(q) had been previously called for this QUEUE. **noenable** instructs **putq** to queue the message when called by this QUEUE, but not to schedule the service procedure. **noenable** does not prevent the QUEUE from being scheduled by a flow control back-enable. The inverse of **noenable** is **enableok**(q).

An example of this is driver upstream flow control. Although device drivers typically discard input when unable to send it to a user process, STREAMS allows driver read side flow control, possibly for handling temporary upstream blocks. This is done through a driver read service procedure which is disabled during the driver open with **noenable.** If the driver input interrupt routine determines messages can be sent upstream (from **canput**), it sends the message with **putnext.** Otherwise, it calls **putq** to queue the message. The message waits on the message queue (possibly with queue length checked when new messages are enqueued by the interrupt routine) until the upstream QUEUE becomes unblocked. When the blockage abates, STREAMS back-enables the driver read service procedure. The service procedure sends the messages upstream using **getq** and **canput**, as in Chapter 8. This is similar to *looprsrv* in Chapter 10 where the service procedure is present only for flow control.

qenable, another flow control utility, allows a module or driver to cause one of its QUEUEs, or another module's QUEUEs, to be scheduled. In addition to the usage shown in Chapters 10 and 11, **qenable** might be used when a module or driver wants to delay message processing for some reason. 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** 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 put procedure calls **qenable** to cause its service procedure to run.

Another example is a communication line discipline module that implements end-to-end (i.e., 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. Then, the read side schedules the write side service procedure to run.

Signals

STREAMS allows modules and drivers to cause a signal to be sent to user process(es) through an M_SIG or M_PCSIG message (see Appendix B) sent upstream. M_PCSIG is a priority version of M_SIG. For both messages, the first byte of the message specifies the signal for the Stream head to generate. If the signal is not SIGPOLL [see *signal*(2) and *sigset*(2)], then the signal is sent to the process group associated with the Stream (see below). If the signal is SIGPOLL, the signal is only sent to processes that have registered for the signal by using the I_SETSIG **ioctl** [also see *streamio*(7)] call.

A process group is associated with a Stream during the open of the driver or module. If *u.u_ttyp* is NULL prior to the driver or module open call, the Stream head checks *u.u_ttyp* after the driver or module open call returns. If *u.u_ttyp* is non-zero, it is assumed to point to a short that holds the process group ID for signaling. The process group and indirect TTY (**/dev/tty**) inode are recorded in the Stream head.

If the driver or module wants to have a process group associated with the Stream, it should include code of the following form in its open procedure:

```
pp = u.u_procp;     /* pointer to process structure */
pdp = ...           /* private data pointer */

if (pp->p_pid == pp->p_pgrp   /* process group leader */
    && u.u_ttyp == NULL       /* with no controlling TTY */
    && pdp->pgrp == 0) {      /* and this stream is unassigned */

        /* assign controlling TTY */

        u.u_ttyp = &pdp->pgrp;
        pdp->pgrp = pp->p_pgrp;
}
```

A private data structure containing a short *pgrp* element is required.

M_SIG can be used by modules or drivers that wish to insert an explicit inband signal into a message stream. For example, an M_SIG message can be sent to the user process immediately before a particular service interface message to gain the immediate attention of the user process. When the M_SIG reaches the head of the Stream head read message queue, a signal will be generated and the M_SIG message will be removed. This leaves the service interface message as the next message to be processed by the user. Use of M_SIG would typically be defined as part of the service interface of the driver or module.

Control of Stream Head Processing

The M_SETOPTS message (see Appendix B) allows a driver or module to exercise control over certain Stream head processing. An M_SETOPTS can be sent upstream at any time. The Stream head responds to the message by altering the processing associated with certain system calls. The options to be modified are specified by the contents of the **stroptions** structure (see Appendix B) contained in the message.

Six Stream head characteristics can be modified. As described in Appendix B, four correspond to fields contained in **queue_t** (min/max packet sizes and high-/low-water marks). The other two are discussed here.

Read Options

The value for read options (*so_readopt*) corresponds to the three modes a user can set via the I_SRDOPT **ioctl** (see *streamio*) call:

byte-stream (RNORM)

The **read**(2) call completes when the byte count is satisfied, the Stream head read queue becomes empty, or a zero-length message is encountered. In the last case, the zero-length message is put back on the queue. A subsequent **read** will return 0 bytes.

message non-discard (RMSGN)

The **read** call completes when the byte count is satisfied or at a message boundary, whichever comes first. Any data remaining in the message is put back on the Stream head read queue.

message discard (RMSGD)

The **read** call completes when the byte count is satisfied or at a message boundary. Any data remaining in the message is discarded.

Byte-stream mode approximately models pipe data transfer. Message non-discard mode approximately models a TTY in canonical mode.

Write Offset

The value for write offset (*so_wroff*) is a hook to allow more efficient data handling. It works as follows: In every data message generated by a **write**(2) system call and in the first M_DATA block of the data portion of every message generated by a **putmsg**(2) call, the Stream head will leave *so_wroff* bytes of space at the beginning of the message block. Expressed as a C language construct:

`bp->b_rptr = bp->b_datap->db_base` +write offset.

The write offset value must be smaller than the maximum STREAMS message size, STRMSGSZ (see the section titled "Tunable Parameters" in Appendix E). In certain cases (e.g., if a buffer large enough to hold the offset+data is not currently available), the write offset might not be included in the block. To be general, modules and drivers should not assume that the offset exists in a message, but should always check the message.

The intended use of write offset is to leave room for a module or a driver to place a protocol header before user data in the message rather than by allocating and prepending a separate message. This feature is not general and its use is discouraged. A more general technique is to put protocol header information in a separate message block and link the user data to it.

A

Appendix A: Kernel Structures

Appendix A: Kernel Structures A-1

streamtab A-1

QUEUE Structures A-2

Message Structures A-4

iocblk A-4

linkblk A-5

Appendix A: Kernel Structures

This appendix summarizes previously described kernel structures commonly encountered in STREAMS module and driver development.

STREAMS kernel structures are contained in **<sys/stream.h>** and **<sys/strstat.h>**.

NOTE These and other STREAMS structures (shown in **bold**) contained in both parts of this guide will remain fixed in subsequent releases of UNIX System V, subject to the following: The offset of all defined elements in each structure will not change. However, the size of the structure may be increased to add new elements.

streamtab

As discussed in Chapter 5, this structure defines a module or driver:

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

QUEUE Structures

Two sets of QUEUE structures form a module. The structures, discussed in Chapters 5 and 8, are **queue_t**, **qinit**, **module_info** and, optionally, **module_stat**:

```
struct queue {
   struct qinit *q_qinfo; /* procedures and limits for queue */
   struct msgb  *q_first; /* head of message queue for this QUEUE */
   struct msgb  *q_last;  /* tail of message queue for this QUEUE */
   struct queue *q_next;  /* next QUEUE in Stream*/
   struct queue *q_link;  /* link to next QUEUE on STREAMS scheduling queue */
   caddr_t       q_ptr;   /* to private data structure */
   ushort        q_count; /* weighted count of characters on message queue */
   ushort        q_flag;  /* QUEUE state */
   short         q_minpsz;/* min packet size accepted by this QUEUE */
   short         q_maxpsz;/* max packet size accepted by this QUEUE */
   ushort        q_hiwat; /* message queue high water mark, for flow control */
   ushort        q_lowat; /* message queue low water mark, for flow control */
};
typedef struct queue queue_t;
```

When a **queue_t** pair is allocated, their contents are zero unless specifically initialized. The following fields are initialized:

- q_qinfo - from streamtab.st_[rd/wr]init (or st_mux[rw]init)
- q_minpsz, q_maxpsz, q_hiwat, q_lowat - from **module_info**
- q_ptr - optionally, by the driver/module open routine

```
struct qinit {
   int   (*qi_putp)();      /* put procedure */
   int   (*qi_srvp)();      /* service procedure */
   int   (*qi_qopen)();     /* called on each open or a push */
   int   (*qi_qclose)();    /* called on last close or a pop */
   int   (*qi_qadmin)();    /* reserved for future use */
   struct module_info *qi_minfo;     /* information structure */
   struct module_stat *qi_mstat;     /* statistics structure - optional */
};
```

```
struct module_info {
   ushort  mi_idnum;     /* module ID number */
   char   *mi_idname;    /* module name */
   short   mi_minpsz;    /* min packet size accepted, for developer use */
   short   mi_maxpsz;    /* max packet size accepted, for developer use */
   short   mi_hiwat;     /* hi-water mark, for flow control */
   ushort  mi_lowat;     /* lo-water mark, for flow control */
};

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

Note that in the event these counts are calculated by modules or drivers, the counts will be cumulative over all occurrences of modules with the same **fmodsw** entry and drivers with the same **cdevsw** entry.

Message Structures

As described in Chapter 7, a message is composed of a linked list of triples, consisting of two structures and a data buffer:

```
struct      msgb {
   struct   msgb   *b_next;     /* next message on queue */
   struct   msgb   *b_prev;     /* previous message on queue */
   struct   msgb   *b_cont;     /* next message block of message */
   unsigned char   *b_rptr;     /* first unread data byte in buffer */
   unsigned char   *b_wptr;     /* first unwritten data byte in buffer  */
   struct   datab  *b_datap;    /* data block */
};
typedef struct msgb mblk_t;

struct datab {
   struct datab   *db_freep; /* used internally */
   unsigned char *db_base;   /* first byte of buffer * */
   unsigned char *db_lim;    /* last byte+1 of buffer */
   unsigned char  db_ref;    /* count of messages pointing to this block */
   unsigned char  db_type;   /* message type */
   unsigned char  db_class;  /* used internally */
};
typedef struct datab dblk_t;
```

iocblk

As described in Chapter 9 and Appendix B, this is contained in an M_IOCTL message block:

```
struct iocblk {
   int    ioc_cmd;    /* ioctl command type */
   ushort ioc_uid;    /* effective uid of user */
   ushort ioc_gid;    /* effective gid of user */
   uint   ioc_id;     /* ioctl id */
   uint   ioc_count;  /* count of bytes in data field */
   int    ioc_error;  /* error code */
   int    ioc_rval;   /* return value  */
};
```

linkblk

As described in Chapter 11, this is used in lower multiplexer drivers:

```
struct linkblk {
   queue_t *l_qtop;   /* lowest level write queue of upper stream */
   queue_t *l_qbot;   /* highest level write queue of lower stream */
   int      l_index;  /* system-unique index for lower stream. */
};
```

B Appendix B: Message Types

Appendix B: Message Types B-1

Ordinary Messages B-2

Priority Messages B-10

Appendix B: Message Types

Eighteen STREAMS message types are defined. The message types differ in their intended purposes, their treatment at the Stream head, and in their message queueing priority (see Chapter 8).

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 are described below, classified according to their message queueing priority. Ordinary messages are described first, with priority messages following. In certain cases, two message types may perform similar functions, differing in priority. Message construction is described in Chapter 7. The use of the word module will generally imply "module or driver."

Ordinary Messages

These message types are subject to flow control. These are referred to as non-priority messages when received at user level.

M_DATA Intended to contain ordinary data. Messages allocated by the *allocb* routine (see Appendix B) are type M_DATA 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** and **putmsg** 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_PROTO Intended to contain internal control information and associated data. The message format is one M_PROTO message block followed by zero or more M_DATA message blocks as shown below: 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 will typically contain 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** and **putmsg** system calls, the control and data parts are passed separately. These calls refer to M_PROTO messages as non-priority messages.

Note that, 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** will compact the blocks into a single control part when passing them to the user process.

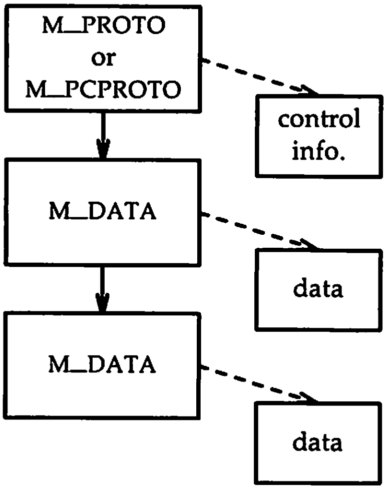

Figure B-1: M_PROTO and M_PCPROTO Message Structure

M_IOCTL Generated by the Stream head in response to an I_STR and certain other **ioctl** system calls [see *streamio*(7)]. When one of these **ioctls** is received from a user process, the Stream head uses values from the process and supplied in the call to create an M_IOCTL message containing them, and sends the message downstream. M_IOCTL messages are intended to perform the general ioctl functions of character device drivers.

The user values are supplied in a structure of the following form, provided as an argument to the **ioctl** call [see I_STR in *streamio*(7)]:

```
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 will wait for acknowledgment to the M_IOCTL message before timing out, *ic_dp* is a pointer to an optional data argument. On input, *ic_len* contains the length of the data argument passed in and, on return from the call, it contains the length of the data, if any, being returned to the user.

The form of an M_IOCTL message is one M_IOCTL message block linked to zero or more M_DATA message blocks. STREAMS constructs an M_IOCTL message block by placing an **iocblk** structure in its data buffer:

```
struct iocblk
{
     int ioc_cmd;         /* ioctl command type */
     ushort ioc_uid;      /* effective user ID number */
     ushort ioc_gid;      /* effective group ID number */
     uint ioc_id;         /* ioctl identifier */
     uint ioc_count;      /* byte count for ioctl data */
     int ioc_error;       /* error code */
     int ioc_rval;        /* return value */
};
```

The **iocblk** structure is defined in **<sys/stream.h>**. *ioc_cmd* corresponds to *ic_cmd*. *ioc_uid* and *ioc_gid* are the effective user and group IDs for the user sending the *ioctl* and can be tested to determine if the user issuing the *ioctl* call is authorized to do so. *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 to match each M_IOCTL message sent downstream with a response which must be sent upstream to the Stream head. The response is contained in an M_IOCACK (positive acknowledgment) or an M_IOCNAK (negative acknowledgment) message. Both these message types have the same format as an M_IOCTL message and contain an **iocblk** structure in the first block with optional data blocks following. If one of these messages reaches the Stream head with an identifier which 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 type into the appropriate response type and set *ioc_count* to 0 if no data is returned. Then, the **qreply** utility (see Appendix C) is used to send the response to the Stream head.

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 may return an error. *ioc_rval* holds any M_IOCACK return value set by a responding module.

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 is no data, *ioc_count* is zero.

If a module wants to send data to a user process as part of its response, it must construct an M_IOCACK message that contains the data. The first message block of this message contains the *iocblk* data structure, with any data stored in one or more M_DATA message blocks linked to the first message block. The module must set *ioc_count* to the number of data bytes sent. On completion of the call, this number is passed to the user in *ic_len*. Data associated with an M_IOCNAK message is not returned to the user process and is discarded by the Stream head.

The first module or a driver that understands the request contained in the M_IOCTL acts on it and generally returns an M_IOCACK message. Intermediate modules that do not recognize a particular request must pass it on. If a driver does not recognize the request, or the receiving module can not acknowledge it, an M_IOCNAK message must be returned.

The Stream head waits for the response message and returns any information contained in an M_IOCACK to the user. The Stream head will "time out" if no response is received in *ic_timeout* interval.

M_CTL	Generated by modules that wish to 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_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_DELAY	Sent to a media driver to request a real-time delay on output. The data buffer associated with this message type 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_PASSFP	This is 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. (A Stream pipe is a Stream that is terminated at both ends by a Stream head; one end of the Stream can always find the other by following the *q_next* pointers in the Stream. The means by which such a structure is created is not described in this document.) The message is generated as a result of an I_SENDFD **ioctl**

[see *streamio*(7)] 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 I_RECVFD **ioctl** [see *streamio*(7)]. The message is placed without passing it through the Stream (i.e., it is not seen by any modules or drivers in the Stream). This message type 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 type, and it can be ignored by module and driver developers.

M_SETOPTS Alters 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:

```
struct stroptions
{
    short so_flags;      /* options to set */
    short so_readopt;    /* read option */
    ushort so_wroff;     /* write offset */
    short so_minpsz;     /* minimum read packet size */
    short so_maxpsz;     /* maximum read packet size */
    ushort so_hiwat;     /* read queue high-water mark */
    ushort so_lowat;     /* read queue low-water mark */
};
```

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)] to RNORM (byte stream), RMSGD (message discard), or RMSGN (message non-discard) as specified by the value of *so_readopt*.

☐ SO_WROFF— Direct the Stream head to insert an offset specified by *so_wroff* into the first message block of all M_DATA messages created as a result of a **write** system call. 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 **mblk_t** structure is greater than *db_base* in the **dblk_t** 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* (see *q_minpsz* in the **queue_t** structure, in Appendix A). 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 0.

☐ SO_MAXPSZ—Change the maximum packet size value associated with the Stream head read queue to *so_maxpsz (see q_maxpsz* in the **queue_t** structure, in Appendix A). 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 on the Stream head read queue to the value specified in *so_hiwat*.

☐ SO_LOWAT— Change the flow control low-water mark (see *q_minpsz* in the **queue_t** structure, Appendix A) on the Stream head read queue to the value specified in *so_lowat*.

M_SIG Sent upstream by modules or drivers to post a signal to a process. When the message reaches the Stream head, the first data byte of the message is transformed into a signal, as defined in **<sys/signal.h>**, to the process(es) according to the following.

If the signal is not SIGPOLL and the Stream containing the sending module or driver is a controlling TTY, the signal is sent to the associated process group. A Stream becomes the controlling TTY for its process group if, on **open**, a module or driver sets *u.u_ttyp* to point to a (short) "process group value."

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*(7)].

Priority Messages

Priority messages are not subject to flow control.

M_PCPROTO	This message type has the same format and characteristics as the M_PROTO message type, except for 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 type is intended to allow data and control information to be sent outside the normal flow control constraints. The **getmsg**(2) and **putmsg**(2) system calls refer to M_PCPROTO messages as priority messages.
M_ERROR	This message type is 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** and **poll**, will fail with *errno* set to the first data byte of the message. POLLERR is set if the Stream is being **polled** [see *poll*(2)]. All processes sleeping on a system call to the Stream are awakened. An M_FLUSH message with an FLUSHRW argument is sent downstream.
M_HANGUP	This message type is 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** and **putmsg** system calls issued to the Stream will fail and return an ENXIO error. Those **ioctls** that cause messages to be sent downstream are also failed. POLLHUP is set if the Stream is being **polled** [see *poll*(2)].

However, subsequent **read** or **getmsg** 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** will return 0, and **getmsg** will set each of its two length fields to 0.

This message also causes a SIGHUP signal to be sent to the process group, if the device is a controlling TTY (see M_SIG).

M_IOCACK This message type signals the positive acknowledgment of a previous M_IOCTL message. The message may contain information sent by the receiving module or driver. The Stream head returns the information to the user if there is a corresponding outstanding M_IOCTL request. The format and use of this message type is described further under M_IOCTL.

M_IOCNAK This message type signals the negative acknowledgment (failure) of a previous M_IOCTL message. When the Stream head receives an M_IOCNAK, the outstanding **ioctl** request, if any, will fail. The format and usage of this message type is described further under M_IOCTL.

M_FLUSH This message type 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 and the write queue of the module.

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 sets the M_FLUSH message flag to FLUSHR, 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 enqueue messages must identify and process this message type.*

M_PCSIG This message type has the same format and characteristics as the M_SIG message type except for priority.

M_START and M_STOP

These messages 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.

C Appendix C: Utilities

Appendix C: Utilities C-1

Utility Descriptions C-3

- **adjmsg** trim bytes in a message C-3
- **allocb** allocate a message block C-3
- **backq** get pointer to the queue behind a given queue C-3
- **bufcall** recover from failure of **allocb** C-4
- **canput** test for room in a queue C-4
- **copyb** copy a message block C-5
- **copymsg** copy a message C-5
- **datamsg** test whether message is a data message C-5
- **dupb** duplicate a message block descriptor C-5
- **dupmsg** duplicate a message C-6
- **enableok** re-allow a queue to be scheduled for service C-6
- **flushq** flush a queue C-6
- **freeb** free a message block C-7
- **freemsg** free all message blocks in a message C-7
- **getq** get a message from a queue C-7
- **insq** put a message at a specific place in a queue C-8
- **linkb** concatenate two messages into one C-8
- **msgdsize** get the number of data bytes in a message C-8
- **noenable** prevent a queue from being scheduled C-9
- **OTHERQ** get pointer to the mate queue C-9
- **pullupmsg** concatenate bytes in a message C-9
- **putbq** return a message to the beginning of a queue C-10

- **putctl** put a control message C-10
- **putctl1** put a control message with a one-byte parameter C-10
- **putnext** put a message to the next queue C-11
- **putq** put a message on a queue C-11
- **qenable** enable a queue C-12
- **qreply** send a message on a stream in the reverse direction C-12
- **qsize** find the number of messages on a queue C-12
- **RD** get pointer to the read queue C-12
- **rmvb** remove a message block from a message C-13
- **rmvq** remove a message from a queue C-13
- **splstr** set processor level C-13
- **strlog** submit messages for logging C-14
- **testb** check for an available buffer C-14
- **unlinkb** remove a message block from the head of a message C-14
- **WR** get pointer to the write queue C-15

Buffer Allocation Priority C-16

Utility Routine Summary C-17

Appendix C: Utilities

This appendix specifies the set of utilities that STREAMS provides to assist development of modules and drivers. There are over 30 utility routines and macros.

The general purpose of the utilities is to perform functions that are commonly used in modules and drivers. However, some utilities also provide the required interrupt environment. A utility must always be used when operating on a message queue and when accessing the buffer pool.

The utilities are contained in either the system source file **io/stream.c** or, if they are macros, in **<sys/stream.h>**.

> NOTE The utilities contained in this appendix represent an interface that will be maintained in subsequent versions of UNIX System V. Other than these utilities (also see the section titled "Accessible Symbols and Functions" in Appendix D), functions contained in the STREAMS kernel code may change between versions.

All structure definitions are contained in Appendix A unless otherwise indicated. All routine references are found in this appendix unless otherwise indicated. The following definitions are used:

Blocked	A queue that cannot be enabled due to flow control (see the section titled "Flow Control" in Chapter 6 of the *Primer*).
Enable	To schedule a queue.
Free	De-allocate a STREAMS storage.
Message block (bp)	A triplet consisting of an **mblk_t** structure, a **dblk_t** structure, and a data buffer. It is referenced by its **mblk_t** structure (see Chapter 7).
Message (mp)	One or more linked message blocks. A message is referenced by its first message block.
Message queue	Zero or more linked messages associated with a queue (**queue_t** structure).

Queue (q)	A **queue_t** structure. This is generally the same as QUEUE in the rest of this document (e.g., see the definitions for enable and schedule). When it appears with "message" in certain utility description lines, it means "message queue".
Schedule	Place a queue on the internal linked list of queues which will subsequently have their service procedure called by the STREAMS scheduler.

The word module will generally mean "module and/or driver". The phrase "next/following module" will generally refer to a module, driver, or Stream head. Message queueing priority (see Chapter 8 and Appendix B) can be ordinary or Priority (to avoid "priority priority").

Utility Descriptions

The utilities are described below. A summary table is contained at the end of this appendix.

adjmsg – trim bytes in a message

int adjmsg(mp, len)
mblk_t *mp;
int len;

adjmsg trims bytes from either the head or tail of the message specified by *mp*. If *len* is greater than zero, it removes *len* bytes from the beginning of *mp*. If *len* is less than zero, it removes (-)*len* bytes from the end of *mp*. If *len* is zero, **adjmsg** does nothing. **adjmsg** only trims bytes across message blocks of the same type. It will fail if *mp* points to a message containing fewer than *len* bytes of similar type at the message position indicated. **adjmsg** returns 1 on success and 0 on failure.

allocb – allocate a message block

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

allocb returns a pointer to a message block of type M_DATA, in which the data buffer contains at least *size* bytes. *pri* indicates the priority of the allocation request and can have the values BPRI_LO, BPRI_MED, or BPRI_HI (see the section titled "Buffer Allocation Priority" in this appendix). If a block can not be allocated as requested, **allocb** returns a NULL pointer.

backq – get pointer to the queue behind a given queue

queue_t *backq(q)
queue_t *q;

backq returns a pointer to the queue behind a given queue. That is, it returns a pointer to the queue whose *q_next* (see *queue_t* structure) pointer is *q*. If no such queue exists (as when *q* is at a Stream end), **backq** returns NULL.

bufcall – recover from failure of **allocb**

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

bufcall is provided to assist in the event of a block allocation failure. If **allocb** returns NULL, indicating a message block is not currently available, **bufcall** may be invoked.

bufcall arranges for (**func*)(*arg*) to be called when a buffer of size bytes at *pri* priority (see the section titled "Buffer Allocation Priority") is available. When *func* is called, it has no user context. It cannot reference the *u_area* and must return without sleeping. **bufcall** does not guarantee that the desired buffer will be available when *func* is called since interrupt processing may acquire it.

bufcall returns 1 on success, indicating that the request has been successfully recorded, or 0 on failure. On a failure return, *func* will never be called. A failure indicates a (temporary) inability to allocate required internal data structures.

canput – test for room in a queue

int canput(q)
queue_t *q;

canput determines if there is room left in a message queue. If *q* does not have a service procedure, **canput** will search further in the same direction in the Stream until it finds a queue containing a service procedure (this is the first queue on which the passed message can actually be enqueued). If such a queue cannot be found, the search terminates on the queue at the end of the Stream. **canput** tests the queue found by the search. If the message queue in this queue is not full (see the section titled "Flow Control" in Chapter 6 of the *Primer*), **canput** returns 1. This return indicates that a message can be put to queue *q*. If the message queue is full, **canput** returns 0. In this case, the caller is generally referred to as blocked.

copyb – copy a message block

mblk_t *copyb(bp)
mblk_t *bp;

copyb copies the contents of the message block pointed to by *bp* into a newly allocated message block of at least the same size. **copyb** allocates a new block by calling **allocb** with *pri* set to BPRI_MED (see the section titled "Buffer Allocation Priority"). All data between the *b_rptr* and *b_wptr* pointers of a message block are copied to the new block, and these pointers in the new block are given the same offset values they had in the original message block. On successful completion, **copyb** returns a pointer to the new message block containing the copied data. Otherwise, it returns a NULL pointer.

copymsg – copy a message

mblk_t *copymsg(mp)
mblk_t *mp;

copymsg uses **copyb** to copy the message blocks contained in the message pointed to by *mp* to newly allocated message blocks, and links the new message blocks to form the new message. On successful completion, **copymsg** returns a pointer to the new message. Otherwise, it returns a NULL pointer.

datamsg – test whether message is a data message

#define datamsg(mp) ...

The **datamsg** macro returns TRUE if *mp* (declared as `mblk_t *mp`) points to a data type message. In this case, types M_DATA, M_PROTO, or M_PCPROTO (see Appendix B). If *mp* points to any other message type, **datamsg** returns FALSE.

dupb – duplicate a message block descriptor

mblk_t *dupb(bp)
mblk_t *bp;

dupb duplicates the message block descriptor (**mblk_t** structure) pointed to by *bp* by copying it into a newly allocated message block descriptor. A message block is formed with the new message block descriptor pointing to the same data block as the original descriptor. The reference count in the data block descriptor (**dblk_t** structure) is incremented. **dupb** does not copy the data buffer, only the message block descriptor.

On successful completion, **dupb** returns a pointer to the new message block. If **dupb** cannot allocate a new message block descriptor, it returns NULL.

This routine allows message blocks that exist on different queues to reference the same data block. In general, if the contents of a message block with a reference count greater than 1 are to be modified, **copyb** should be used to create a new message block and only the new message block should be modified. This insures that other references to the original message block are not invalidated by unwanted changes.

dupmsg – duplicate a message

mblk_t *dupmsg(mp)
mblk_t *mp;

dupmsg calls **dupb** to duplicate the message pointed to by *mp*, by copying all individual message block descriptors, and then linking the new message blocks to form the new message. **dupmsg** does **not** copy data buffers, only message block descriptors. On successful completion, **dupmsg** returns a pointer to the new message. Otherwise, it returns NULL.

enableok – re-allow a queue to be scheduled for service

#define enableok(q) ...

The **enableok** macro cancels the effect of an earlier **noenable** on the same queue *q* (declared as queue_t *q). It allows a queue to be scheduled for service that had previously been excluded from queue service by a call to **noenable**.

flushq – flush a queue

int flushq(q, flag)
queue_t *q;
int flag;

flushq removes messages from the message queue in queue *q* and frees them, using **freemsg**. If *flag* is set to FLUSHDATA, then **flushq** discards all M_DATA, M_PROTO, and M_PCPROTO messages (see *datamsg*), but leaves all other messages on the queue. If *flag* is set to FLUSHALL, all messages are removed from the message queue and freed. FLUSHALL and FLUSHDATA are defined in **<sys/stream.h>**.

If a queue behind *q* is blocked, **flushq** may enable the blocked queue, as described in **putq**.

freeb – free a message block

int freeb(bp)
mblk_t *bp;

freeb will free (deallocate) the message block descriptor pointed to by *bp*, and will free the corresponding data block if the reference count (see *dupb*) in the data block descriptor (**dblk_t** structure) is equal to 1. If the reference count is greater than 1, **freeb** will not free the data block, but will decrement the reference count.

freemsg – free all message blocks in a message

int freemsg(mp)
mblk_t *mp;

freemsg uses **freeb** to free all message blocks and their corresponding data blocks for the message pointed to by *mp*.

getq – get a message from a queue

mblk_t *getq(q)
queue_t *q;

getq gets the next available message from the queue pointed to by *q*. **getq** returns a pointer to the message and removes that message from the queue. If no message is queued, **getq** returns NULL.

getq and certain other utility routines affect flow control in the Stream as follows: If **getq** returns NULL, the queue is internally marked so that the next time a message is placed on it, it will be scheduled for service (enabled, see *qenable*). Also, if the data in the enqueued messages in the queue drops below the low-water mark, *q_lowat*, and a queue behind the current queue had previously attempted to.place a message in the queue and failed (i.e., was blocked, see *canput*), then the queue behind the current queue is scheduled for service (see the section titled "Flow Control" in Chapter 6 of the *Primer*).

insq – put a message at a specific place in a queue

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

insq places the message pointed to by *nmp* in the message queue contained in the queue pointed to by *q* immediately before the already enqueued message pointed to by *emp*. If *emp* is NULL, the message is placed at the end of the queue. If *emp* is non-NULL, it must point to a message that exists on the queue *q*, or a system panic could result.

Note that the message is placed where indicated, without consideration of message queueing priority. The queue will be scheduled in accordance with the rules described in **putq** for ordinary priority messages.

linkb – concatenate two messages into one

int linkb(mp1, mp2)
mblk_t *mp1;
mblk_t *mp2;

linkb puts the message pointed to by *mp2* at the tail of the message pointed to by *mp1*.

msgdsize – get the number of data bytes in a message

int msgdsize(mp)
mblk_t *mp;

msgdsize returns the number of bytes of data in the message pointed to by *mp*. Only bytes included in data blocks of type M_DATA are included in the total.

noenable – prevent a queue from being scheduled

#define noenable(q) ....

The **noenable** macro prevents the queue *q* (declared as queue_t *q) from being scheduled for service by **putq** or **putbq** when these routines enqueue an ordinary priority message, or by **insq** when it enqueues any message. **noenable** does not prevent the scheduling of queues when a Priority message is enqueued, unless it is enqueued by **insq**.

OTHERQ – get pointer to the mate queue

#define OTHERQ(q) ...

The **OTHERQ** macro returns a pointer to the mate queue of *q* (declared as queue_t *q). If *q* is the read queue for the module, it returns a pointer to the module's write queue. If *q* is the write queue for the module, it returns a pointer to the read queue.

pullupmsg – concatenate bytes in a message

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

pullupmsg concatenates and aligns the first *len* data bytes of the passed message into a single, contiguous message block. Proper alignment is hardware-dependent. To perform its function, **pullupmsg** allocates a new message block by calling **allocb** with *pri* set to BPRI_MED (see the section titled "Buffer Allocation Priority"). **pullupmsg** only concatenates across message blocks of similar type. It will fail if *mp* points to a message of less than *len* bytes of similar type. A *len* value of -1 requests a pull-up of all the like-type blocks in the beginning of the message pointed to by *mp*.

At completion of concatenation, **pullupmsg** replaces *mp* with a pointer to the new message block, so that *mp* still points to the same message block at the end of the operation. However, the contents of the message block may have been altered. On success, **pullupmsg** returns 1. On failure, it returns 0.

putbq – return a message to the beginning of a queue

int putbq(q, bp)
queue_t *q;
mblk_t *bp

putbq puts the message pointed to by *bp* at the beginning of the queue pointed to by *q*, in a position in accordance with the message's type. Priority messages are placed at the head of the queue, and ordinary messages are placed after all Priority messages, but before all other ordinary messages. The queue will be scheduled in accordance with the same rules described in **putq**. This utility is typically used to replace a message on a queue from which it was just removed.

putctl – put a control message

int putctl(q, type)
queue_t *q;
int type;

putctl creates a control (not data, see *datamsg* above) message of type *type*, and calls the *put* procedure in the queue pointed to by *q*, with a pointer to the created message as an argument. **putctl** allocates new blocks by calling **allocb** with *pri* set to BPRI_HI (see the section titled "Buffer Allocation Priority"). On successful completion, **putctl** returns 1. It returns 0 if it cannot allocate a message block, or if *type* M_DATA, M_PROTO, or M_PCPROTO was specified.

putctl1 – put a control message with a one-byte parameter

int putctl1(q, type, p)
queue_t *q;
int type;
int p;

putctl1 creates a control (not data, see *datamsg*) message of type *type* with a one-byte parameter *p*, and calls the *put* procedure in the queue pointed to by *q*, with a pointer to the created message as an argument. **putctl1** allocates new blocks by calling **allocb** with *pri* set to BPRI_HI (see the section titled "Buffer Allocation Priority"). On successful completion, **putctl1** returns 1. It returns 0 if it cannot allocate a message block, or if *type* M_DATA, M_PROTO, or M_PCPROTO was specified.

putnext – put a message to the next queue

#define putnext(q, mp) ...

The **putnext** macro calls the *put* procedure of the next queue in a Stream, and passes it a message pointer as an argument. The parameters must be declared as `queue_t *q` and `mblk_t *mp`. *q* is the calling queue (not the next queue) and *mp* is the message to be passed. **putnext** is the typical means of passing messages to the next queue in a Stream.

putq – put a message on a queue

int putq(q, bp)
queue_t *q;
mblk_t *bp;

putq puts the message pointed to by *bp* on the message queue contained in the queue pointed to by *q* and enables that queue. **putq** queues messages appropriately by type (i.e., message queueing priority, see Chapter 8).

putq will always enable the queue when a Priority message is queued. **putq** will enable the queue when an ordinary message is queued if the following condition is set, and enabling is not inhibited by **noenable**: The condition is set if the module has just been pushed [see I_PUSH in *streamio*(7)], or if no message was queued on the last **getq** call, and no message has been queued since.

putq is intended to be used from the put procedure in the same queue in which the message will be queued. A module should not call **putq** directly to pass messages to a neighboring module. **putq** may be used as the *qi_putp*() put procedure value in either or both of a module's **qinit** structures. This effectively bypasses any put procedure processing and uses only the module's service procedure(s).

qenable – enable a queue

int qenable(q) queue_t *q;

int putq(q, bp)
queue_t *q;
mblk_t *bp;

qenable places the queue pointed to by *q* on the linked list of queues that are ready to be called by the STREAMS scheduler (see the definition for "Schedule" above, and the section titled "Put and Service Procedures" in Chapter 5 of the *Primer*).

qreply – send a message on a stream in the reverse direction

int qreply(q, bp)
queue_t *q;
mblk_t *bp;

qreply sends the message pointed to by *bp* up (or down) the Stream in the reverse direction from the queue pointed to by *q*. This is done by locating the partner of *q* (see OTHERQ) and then calling the *put* procedure of that queue's neighbor (as in **putnext**). **qreply** is typically used to send back a response (M_IOCACK or M_IOCNAK message) to an M_IOCTL message (see Appendix B).

qsize – find the number of messages on a queue

int qsize(q)
queue_t *q;

qsize returns the number of messages present in queue *q*. If there are no messages on the queue, **qsize** returns 0.

RD – get pointer to the read queue

#define RD(q) ...

The **RD** macro accepts a write queue pointer, *q* (declared as `queue_t *q`), as an argument and returns a pointer to the read queue for the same module.

rmvb – remove a message block from a message

mblk_t *rmvb(mp, bp)
mblk_t *mp;
mblk_t *bp;

rmvb removes the message block pointed to by *bp* from the message pointed to by *mp* and then restores the linkage of the message blocks remaining in the message. **rmvb** does not free the removed message block. **rmvb** returns a pointer to the head of the resulting message. If *bp* is not contained in *mp,* **rmvb** returns a -1. If there are no message blocks in the resulting message, **rmvb** returns a NULL pointer.

rmvq – remove a message from a queue

int rmvq(q, mp)
queue_t *q;
mblk_t *mp;

rmvq removes the message pointed to by *mp* from the message queue in the queue pointed to by *q* and then restores the linkage of the messages remaining on the queue. If *mp* does not point to a message that is present on the queue *q,* a system panic could result.

splstr – set processor level

int splstr()

splstr increases the system processor level to block interrupts at a level appropriate for STREAMS modules when those modules are executing critical portions of their code. **splstr** returns the processor level at the time of its invocation. Module developers are expected to use the standard kernel function **splx(s),** where *s* is the integer value returned by **splstr,** to restore the processor level to its previous value after the critical portions of code are passed.

strlog – submit messages for logging

int strlog(mid, sid, level, flags, fmt, arg1, ...)
short mid, sid;
char level;
ushort flags;
char *fmt;
unsigned arg1;

strlog submits messages containing specified information to the **log** driver. Required definitions are contained in **<sys/strlog.h>** and **<sys/log.h>**. *mid* is the STREAMS module ID number for the module or driver submitting the **log** message. *sid* is an internal sub-ID number usually used to identify a particular minor device of a driver. *level* is a tracing level that allows selective screening of messages from the tracer. *flags* are 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), and SL_NOTIFY (request that a copy of the message be mailed to the system administrator). *fmt* is a **printf**(3S) style format string, except that %s, %e, %E, %g, and %G conversion specifications are not handled. Up to NLOGARGS numeric or character arguments can be provided. [See Chapter 6 of the *Primer* and *log*(7).]

testb – check for an available buffer

int testb(size, pri)
int size, pri;

testb checks for the availability of a message buffer of size *size* at priority *pri* (see the section titled "Buffer Allocation Priority") without actually retrieving the buffer. **testb** returns 1 if the buffer is available and 0 if no buffer is available. A successful return value from **testb** does not guarantee that a subsequent **allocb** call will succeed (e.g., in the case of an interrupt routine taking buffers).

unlinkb – remove a message block from the head of a message

mblk_t *unlinkb(mp)
mblk_t *mp;

unlinkb removes the first message block pointed to by *mp* and returns a pointer to the head of the resulting message. **unlinkb** returns a NULL pointer if there are no more message blocks in the message.

WR – get pointer to the write queue

#define WR(q) ...

The **WR** macro accepts a read queue pointer, *q* (declared as `queue_t *q`), as an argument and returns a pointer to the write queue for the same module.

Buffer Allocation Priority

STREAMS buffers are normally allocated with **allocb**, described above. An associated set of allocation priorities has been established, which are also used in other utility routines:

BPRI_LO	Low priority. At this priority, **allocb** may fail even though the requested buffer size is available. This priority is used by the Stream head write routine to hold data associated with user calls.
BPRI_MED	Medium priority. This priority is typically used for normal data and control block allocation. As above, **allocb** may fail at this priority even though a buffer of the requested size is available. However, for a given block size, an BPRI_LO **allocb** call will fail before a BPRI_MED **allocb** call.
BPRI_HI	High priority. This priority is typically used only for critical control message allocations. Calls to **allocb** will succeed if a buffer of the appropriate size is available. Developers should exercise restraint in use of BPRI_HI allocation requests.

The values BPRI_LO, BPRI_MED, and BPRI_HI are defined in **<sys/stream.h>**.

STREAMS does not guarantee successful buffer allocation—any set of resources can be exhausted under the right conditions. The **bufcall** function will help modules recover from buffer allocation failures, but it does not guarantee that the resources will ever be available. Developers should be aware of this when implementing modules.

Utility Routine Summary

ROUTINE	DESCRIPTION
adjmsg	trim bytes in a message
allocb	allocate a message block
backq	get pointer to the queue behind a given queue
bufcall	recover from failure of **allocb**
canput	test for room in a queue
copyb	copy a message block
copymsg	copy a message
datamsg	test whether message is a data message
dupb	duplicate a message block descriptor
dupmsg	duplicate a message
enableok	re-allow a queue to be scheduled for service
flushq	flush a queue
freeb	free a message block
freemsg	free all message blocks in a message
getq	get a message from a queue
insq	put a message at a specific place in a queue
linkb	concatenate two messages into one
msgdsize	get the number of data bytes in a message
noenable	prevent a queue from being scheduled
OTHERQ	get pointer to the mate queue
pullupmsg	concatenate bytes in a message
putbq	return a message to the beginning of a queue
putctl	put a control message
putctl1	put a control message with a one-byte parameter
putnext	put a message to the next queue
putq	put a message on a queue
qenable	enable a queue
qreply	send a message on a stream in the reverse direction
qsize	find the number of messages on a queue
RD	get pointer to the read queue
rmvb	remove a message block from a message
rmvq	remove a message from a queue
splstr	set processor level
strlog	submit messages for logging
testb	check for an available buffer
unlinkb	remove a message block from the head of a message
WR	get pointer to the write queue

D

Appendix D: Design Guidelines

Appendix D: Design Guidelines D-1
General Rules D-1
System Calls D-2
Data Structures D-3
Header Files D-3
Accessible Symbols and Functions D-4
Rules for Put and Service Procedures D-5
Error and Trace Logging D-7

Appendix D: Design Guidelines

This appendix summarizes STREAMS module and driver design guidelines and rules presented in previous chapters. Additional rules that developers must observe are included. Where appropriate, the section of this document containing detailed information is named. The end of the appendix contains a brief description of error and trace logging facilities.

Unless otherwise noted, "module" implies "modules and drivers."

General Rules

The following are general rules that developers should follow when writing modules.

1. Modules cannot access information in the **u_area** of a process. Modules are not associated with any process, and therefore, have no concept of process or user context.

   The capability to pass **u_area** information upstream using messages has been provided where required. This can be done in M_IOCTL handling (see Chapter 9 and Appendix B). A module can send error codes upstream in an M_IOCACK or M_IOCNAK message, where they will be placed in *u_error* by the Stream head. Return values may also be sent upstream in a M_IOCACK message and will be placed in *u_rval1*. Information can also be passed to the **u_area** via a M_ERROR message (see Chapter 10 and Appendix B). The Stream head will recognize this message type and inform the next system call that an error has occurred downstream by setting *u_error*. Note that in both instances, the downstream module cannot access the **u_area**, but it informs the Stream head to do so.
2. 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 may limit module reusability (the ability to use the module in multiple applications).
3. Every module must process an M_FLUSH message according to the value of the argument passed in the message (see Chapters 8 and 9, and Appendix B).

4. A module should not change the contents of a data block whose reference count is greater than 1 (see *dupmsg* in Appendix C) because other modules that have references to the block may not want the data changed. To avoid problems, it is recommended that the module copy the data to a new block and then change the new one.
5. Modules should only manipulate message queues and manage buffers with the routines provided for those purpose (see Appendix C).
6. Filter modules pushed between a service user and a service provider (see Chapter 12) may not alter the contents of the M_PROTO or M_PCPROTO block in messages. The contents of the data blocks may be manipulated, but the message boundaries must be preserved.

System Calls

These rules pertain to module and drivers as noted.

1. *open* and *close* routines may sleep, but the sleep must return to the routine in the event of a signal. That is, if they sleep, they must be at priority <= PZERO or with PCATCH set in the sleep priority.
2. The *open* routine must return >= zero on success or OPENFAIL if it fails. This ensures that a failure will be reported to the user process. *errno* may be set on failure. However, if the open routine returns OPENFAIL and *errno* is not set, STREAMS will automatically set *errno* to ENXIO.
3. If a module or driver recognizes and acts on an M_IOCTL message, it must reply by sending a M_IOCACK message upstream. A unique ID is associated with each M_IOCTL, and the M_IOCACK or M_IOCNAK message must contain the ID of the M_IOCTL it is acknowledging.
4. A module (not a driver) must pass on any M_IOCTL message it does not recognize (see Appendix B). If an unrecognized M_IOCTL reaches a driver, the driver must reply by sending a M_IOCNAK message upstream.

Data Structures

Only the contents of *q_ptr, q_minpsz, q_maxpsz, q_hiwat,* and *q_lowat* in a **queue_t** structure may be altered. The latter four quantities are set when the module or driver is opened, but may be modified subsequently.

As described in Appendix E, every module and driver is configured with the address of a **streamtab** structure (see Chapter 5). For a driver, a pointer to its **streamtab** is included in **cdevsw**. For a module, a pointer to its **streamtab** is included in **fmodsw**.

Header Files

The following header files are generally required in modules and drivers:

types.h	contains type definitions used in the STREAMS header files
stream.h	contains required structure and constant definitions
stropts.h	primarily for users, but contains definitions of the arguments to the M_FLUSH message type also required by modules

One or more of the header files described below may also be included (also see the following section). No standard UNIX System header files should be included except as described in the following section. The intent is to prevent attempts to access data that cannot or should not be accessed.

errno.h	defines various system error conditions and is needed if errors are to be returned upstream to the user
sysmacros.h	contains miscellaneous system macro definitions
param.h	defines various system parameters, particularly the value of the PCATCH sleep flag
signal.h	defines the system signal values and should be used if signals are to be processed or sent upstream
file.h	defines the file open flags and is needed if O_NDELAY is interpreted

Accessible Symbols and Functions

The following lists the only symbols and functions that modules or drivers may refer to (in addition to those defined by STREAMS), if hardware and UNIX System release independence is to be maintained. Use of symbols not listed here is unsupported.

- **user.h** (from open/close procedures only)

struct proc *u_procp	process structure pointer
short *u_ttyp	TTY group ID pointer
char u_error	system call error number
ushort u_uid	effective user ID
ushort u_gid	effective group ID
ushort u_ruid	real user ID
ushort u_rgid	real group ID

- **proc.h** (from open/close procedures only)

short p_pid	process ID
short p_pgrp	process group ID

- **functions accessible from open/close procedures only**

flg = sleep(chan, pri)	sleep until wakeup
delay(ticks)	delay for a specified time

- **universally accessible functions**

bcopy(from, to, nbytes)	copy data quickly
bzero(buffer, nbytes)	zero data quickly
t = max(a, b)	return max of args
t = min(a, b)	return min of args
mem=malloc(mp, size)	allocate memory space
mfree(mp, size, i)	deallocate memory space
mapinit(mp, mapsize)	initialize map structure
addr = vtop(vaddr, NULL)	translate from virtual to physical address
printf(format, ...)	print message
cmn_err(level, ...)	print message and optional panic
s = spl*n*()	set priority level
id = timeout(func, arg, ticks)	schedule event
untimeout(id)	cancel event
wakeup(chan)	wake up sleeper

- **sysmacros.h**

t = major(dev)	return major device
t = minor(dev)	return minor device

- **systm.h**

time_t lbolt	clock ticks since boot in HZ
time_t time	seconds since epoch

- **param.h**

PZERO	zero sleep priority
PCATCH	catch signal sleep flag
HZ	clock ticks per second
NULL	0

- **types.h**

dev_t	combined major/minor device
time_t	time counter

All data elements are software read-only except:

u_error - may be set on a failure return of open
u_ttyp - may be set in open to create a controlling TTY

Rules for Put and Service Procedures

To ensure proper data flow between modules, the following rules should be observed in put and service procedures. The following rules pertain to put procedures.

1. A put procedure must not sleep.
2. Each QUEUE must define a put procedure in its **qinit** (see Appendix A) structure for passing messages between modules.
3. A put procedure must use the **putq** (see Appendix C) utility to enqueue a message on its own message queue. This is necessary to ensure that the various fields of the **queue_t** structure are maintained consistently.
4. When passing messages to a neighbor module, a module may not call **putq** directly, but must call its neighbor's put procedure (see *putnext* in Appendix C). Note that this rule is distinct from the one above it. The previous rule states that a module must call **putq** to place

messages on its own message queue, whereas this rule states that a module must not call **putq** directly to place messages on a neighbor's queue.

However, the **q_qinfo** structure that points to a module's put procedure may point to **putq** (i.e. **putq** is used as the put procedure for that module). When a module calls a neighbor's put procedure that is defined in this manner, it will be calling **putq** indirectly. If any module uses **putq** as its put procedure in this manner, the module must define a service procedure. Otherwise, no messages will ever be sent to the next module. Also, because **putq** does not process M_FLUSH messages, any module that uses **putq** as its put procedure must define a service procedure to process M_FLUSH messages.

5. The put procedure of a QUEUE with no service procedure must call the put procedure of the next QUEUE directly if a message is to be passed to that QUEUE. If flow control is desired, a service procedure must be provided.

Service procedures must observe the following rules:

1. A service procedure must not sleep.
2. The service procedure must use **getq** to remove a message from its message queue so that the flow control mechanism is maintained.
3. The service procedure should process all messages on its message queue. The only exception is if the Stream ahead is blocked (i.e., **canput** fails, see Appendix C). Adherence to this rule is the only guarantee that STREAMS will enable (schedule for execution) the service procedure when necessary, and that the flow control mechanism will not fail.

   If a service procedure exits for any other reason (e.g., buffer allocation failure), it must take explicit steps to assure it will be re-enabled.
4. The service procedure must follow the steps below for each message that it processes. STREAMS flow control relies on strict adherence to these steps.

Step 1: Remove the next message from the message queue using **getq**. It is possible that the service procedure could be called when no messages exist on the queue, so the service procedure should never assume that there is a message on its message queue. If there is no message, return.

Step 2: If all the following conditions are met:

- ☐ **canput** fails and
- ☐ the message type is not a priority type (see Appendix B) and
- ☐ the message is to be put on the next QUEUE.

then, continue at Step 3. Otherwise, continue at Step 4.

Step 3: The message must be replaced on the head of the message queue from which it was removed using **putbq** (see Appendix C). Following this, the service procedure is exited. The service procedure should not be re-enabled at this point. It will be automatically back-enabled by flow control.

Step 4: If all the conditions of Step 2 are not met, the message should not be returned to the queue. It should be processed as necessary. Then, return to Step 1.

Error and Trace Logging

STREAMS error and trace loggers are provided for debugging and for administering modules and driver. Chapter 6 of the *STREAMS Primer* contains a description of this facility which consists of **log**, **strace**, **strclean**, **strerr**, and the **strlog** function described in Appendix C.

E Appendix E: Configuring

Appendix E: Configuring E-1
Configuring STREAMS Modules and Drivers E-1
- Configuration Mechanism E-2
- Configuration Examples E-3

Tunable Parameters E-4
System Error Messages E-6

Appendix E: Configuring

This appendix contains information about configuring STREAMS modules and drivers into UNIX System V/386 Release 3.2 on your computer. The information is incremental and presumes the reader is familiar with the configuration mechanism, which may vary on different processors. An example of how to configure a driver and a module is included.

This appendix also includes a list of STREAMS system tunable parameters and system error messages.

Configuring STREAMS Modules and Drivers

Each character device that is configured into a UNIX System results in an entry being placed in the kernel **cdevsw** table. Entries for STREAMS drivers are also placed in this table. However, because system calls to STREAMS drivers must be processed by the STREAMS routines, the configuration mechanism distinguishes between STREAMS drivers and character device drivers in their associated **cdevsw** entries.

The distinction is contained in the *d_str* field which was added to the **cdevsw** structure for this purpose. *d_str* provides the appropriate single entry point for all system calls on STREAMS files, as shown below:

```
extern  struct  cdevsw  {
    .
    .
    .
    struct streamtab *d_str;
} cdevsw[ ];
```

The configuration mechanism forms the *d_str* entry name by appending the string "info" to the STREAMS driver prefix. The "info" entry is a pointer to a **streamtab** structure (see Appendix A) that contains pointers to the **qinit** structures for the read and write QUEUEs of the driver. The driver must contain the external definition:

`struct streamtab` *prefix*`info = { ...`

If the *d_str* entry contains a non-NULL pointer, the operating system will recognize the device as a STREAMS driver and will call the appropriate STREAMS routine. If the entry is NULL, a character I/O device **cdevsw** interface is used. Note that only **streamtab** must be externally defined in STREAMS drivers and modules. **streamtab** is used to identify the appropriate open, close, put, service, and administration routines. These driver/module routines should generally be declared `static`.

The configuration mechanism supports various combinations of block, character, STREAMS devices, and STREAMS modules (see below). For example, it is possible to identify a device as a block *and* STREAMS device, and entries will be inserted in the appropriate system switch tables. A device cannot be both a character and STREAMS device.

When a STREAMS module is configured, an **fmodsw** table entry is generated by the configuration mechanism. **fmodsw** contains the following:

```
#define FMNAMESZ 8

extern struct fmodsw {
      char f_name[FMNAMESZ+1];
      struct streamtab *f_str;
} fmodsw[ ];
```

f_name is the name of the module used in STREAMS-related **ioctl** calls. *f_str* is similar to the *d_str* entry in the **cdevsw** table. It is a pointer to a **streamtab** structure which contains pointers to the **qinit** structures for the read and write QUEUEs of this STREAMS module (as in STREAMS drivers). The module must contain the external definition:

`struct streamtab` *prefix*`info = { ...`

Configuration Mechanism

STREAMS modules and drivers are configured into the system by the following (see the *Operations/System Administrator's Guide* for further details):

1. Use **idcheck** command to determine if the driver is already installed, and to verify that the interrupt vector, I/O address, or other selectable parameter is in fact available for use.
2. Prepare **Master, System, Driver.o** (mandatory) and **Space.c, Node, Init, Rc, Shutdown** (optional), and use **idinstall** command to install them.
3. Use **idtune** command to add tunable parameters to **mtune** or **stune** files.

ID tools are in **/etc/conf/bin**.

NOTE

To specify a STREAMS device driver, its Master file should contain both an **S** and a **c** in the characteristics field (the third field).

A STREAM module that is not a device driver requires an **S** in the characteristics field in its Master file, but should not include a **c** as a device driver does.

Any combination of block, STREAMS drivers, and STREAMS module may be specified. However, it is illegal to specify a STREAMS device or module with a character device.

Configuration Examples

This section contains examples of configuring the following STREAMS driver and module:

loop	the STREAMS loop-around software driver of Chapter 10
crmod	the conversion module of Chapter 7

To configure the STREAMS software (pseudo-device) driver, *loop*, the following three files are necessary:

Master File

```
loop    -    Sioc    loop    0    0    1    1    -1
```

1. **System File**

```
loop    Y    1    0    0    0    0    0    0    0
```

2. **Space.c**

```
#define NLP 2
struct loop loop_loop[NLP];
int loop_cnt = NLP;
```

To configure the STREAMS module **crmod**, the following two files are necessary:

Master File

```
crmd     -     Sio     crmd     0     0     1     1     -1
```

1. **System File**

```
crmd     Y     1     0     0     0     0     0     0     0
```

To configure **crmod** and loop into a new kernel, run **idinstall**.

To move or append driver component files to files in the **/etc/conf** directory, run the **idbuild** command.

Tunable Parameters

Certain system parameters referenced by STREAMS are configurable when building a new operating system (see the *Operations/System Administrator's Guide* for further details). This can be done by including the appropriate entry in the kernel master file. "queues" refers to **queue_t** structures. These parameters are:

NQUEUE	Total number of queues that may be allocated at one time by the system. Queues are allocated in pairs. Each STREAMS driver, Stream head, and pushable module requires a pair of queues. A minimal Stream contains 4 queues (two for the Stream head, two for the driver).
NSTREAM	Total number of Streams that may be open at one time in a system.
NBLK4096	Total number of 4096-byte data blocks available for STREAMS operations. The pool of data blocks is a system-wide resource, so enough blocks must be configured to satisfy all Streams.
NBLK2048	Total number of 2048-byte data blocks available for STREAMS operations.

NBLK1024	Total number of 1024-byte data blocks available for STREAMS operations.
NBLK512	Total number of 512-byte data blocks available for STREAMS operations.
NBLK256	Total number of 256-byte data blocks available for STREAMS operations.
NBLK128	Total number of 128-byte data blocks available for STREAMS operations.
NBLK64	Total number of 64-byte data blocks available for STREAMS operations.
NBLK16	Total number of 16-byte data blocks available for STREAMS operations.
NBLK4	Total number of 4-byte data blocks available for STREAMS operations.
NMUXLINK	Total number of Streams in the system that can be linked as lower Streams to multiplexer drivers [by an I_LINK **ioctl**, see *streamio*(7)].
NSTREVENT	Initial number of internal event cells available in the system to support **bufcall** (see Appendix C) and **poll** [see *poll*(2)] calls.
MAXSEPGCNT	The number of additional pages of memory that can be dynamically allocated for event cells. If this value is 0, only the allocation defined by NSTREVENT is available for use. If the value is not 0 and if the kernel runs out of event cells, it will under some circumstances attempt to allocate an extra page of memory from which new event cells can be created. MAXSEPGCNT places a limit on the number of pages that can be allocated for this purpose. Once a page has been allocated for event cells, however, it cannot be recovered later for use elsewhere.
NSTRPUSH	Maximum number of modules that may be pushed onto a single Stream.
STRMSGSZ	Maximum 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** exceeding

	this size will be broken into multiple messages. A **putmsg** with a data part exceeding this size will fail.
STRCTLSZ	Maximum 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** with a control part exceeding this size will fail.
STRLOFRAC	The percentage of data blocks of a given class at which low priority block allocation requests are automatically failed. For example, if STRLOFRAC is 80 and there are forty-eight 256-byte blocks, a low priority allocation request will fail when more than thirty-eight 256-byte blocks are already allocated. This value is used to prevent deadlock situations in which a low priority activity might starve out more important functions. For example, if STRLOFRAC is 80 and there are 100 blocks of 256 bytes, then when more than 80 of such blocks are allocated, any low priority allocation request will fail. This value must be in the range 0 <= STRLOFRAC <= STRMEDFRAC.
STRMEDFRAC	The percentage of data blocks of a given class at which medium priority block allocation requests are automatically failed.

System Error Messages

Messages are reported to the console as a result of various error conditions detected by STREAMS. These messages and the action to be taken on their occurrence are described below. In certain cases, a tunable parameter (see previous section) may have to be changed.

stropen: out of streams

A Stream head data structure could not be allocated during the **open** of a STREAMS device. If this occurs repeatedly, increase NSTREAM.

stropen: out of queues

A pair of queues could not be allocated for the Stream head during the **open** of a driver. If this occurs repeatedly, increase NQUEUE.

KERNEL: allocq: out of queues

A pair of queues could not be allocated for a pushable module (I_PUSH **ioctl**) or driver (**open**). If this occurs repeatedly, increase NQUEUE.

strinit: can not allocate stream data blocks

During system initialization, the system was unable to allocate enough memory for the STREAMS data blocks. The system must be rebuilt with fewer data blocks specified.

KERNEL: strinit: odd value configured for v.v_nqueue
KERNEL: strinit: was *qcnt*, set to *nqcnt*

During system initialization, the total number of queues allocated, *qcnt*, was not a multiple of 2. The system resets this to an appropriate value, *nqcnt*.

WARNING: bufcall: could not allocate stream event

A call to **bufcall** has failed because all Stream event cells have been allocated. If this occurs repeatedly, increase NSTREVENT.

KERNEL: sealloc: not enough memory for page allocation

An attempt to dynamically allocate a page of Stream event cells failed. If this occurs repeatedly, decrease MAXSEPGCNT.

KERNEL: munlink: could not perform ioctl, closing anyway

A linked multiplexer could not be unlinked when the controlling Stream for that link was closed. The linked Stream will be unlinked and the controlling Stream will be closed anyway.

Glossary

Back-enable	To enable (by STREAMS) a preceding blocked QUEUE when STREAMS determines that a succeeding QUEUE has reached its low-*water mark.*
Blocked	A QUEUE that cannot be enabled due to *flow control.*
Clone device	A STREAMS device that returns an unused minor device when initially opened, rather than requiring the minor device to be specified in the **open**(2) call.
Close procedure	The module routine that is called when a module is popped from a Stream and the driver routine that is called when a *driver* is closed.
Control stream	In a *multiplexer,* the *upper Stream* on which a previous I_LINK **ioctl** [to the associated file, see *streamio*(7)] caused a *lower Stream* to be connected to the multiplexer driver at the end of the *upper Stream.*
Downstream	The direction from *Stream head* towards *driver.*
Device driver	The end of the *Stream* closest to an external interface. The principle functions of a *device driver* 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.* In STREAMS, a *driver* is physically identical to a module (i.e., composed of two QUEUEs), but has additional attributes in a Stream and in the UNIX system.
Enable	Schedule a QUEUE.
Flow control	The STREAMS mechanism that regulates the flow of messages within a Stream and the flow from user space into a Stream.
Lower Stream	A Stream connected below a multiplexer *pseudo-device driver,* by means of an I_LINK **ioctl**. The far end of a lower Stream terminates at a *device driver* or another multiplexer driver.

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	Carries data or information, as identified by its *message type*, in a Stream. A *message block* is a triplet consisting of a data buffer and associated control structures, an **mblk_t** structure, and a **dblk_t** structure.
Message queue	A linked list of zero or more *messages* connected to a QUEUE.
Message type	A defined set of values identifying the contents of a *message block* and *message.*
Module	A pair of QUEUEs. In general, module implies a *pushable* module.
Multiplexer	A STREAMS mechanism that allows messages to be routed among multiple Streams in the kernel. A multiplexer includes at least one multiplexing *pseudo-device driver* connected to one or more *upper* Streams and one or more *lower* Streams.
Open procedure	The routine 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.
Pop	A STREAMS **ioctl** [see *streamio*(7)] that causes the *pushable module* immediately below the *Stream head* to be removed (popped) from a Stream [modules can also be popped as the result of a **close**(2)].
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.*
Push	A STREAMS **ioctl** [see *streamio*(7)] that causes a *pushable module* to be inserted (pushed) in a Stream immediately below the *Stream head.*

Pushable module	A module interposed (pushed) between the *Stream head* and *driver*. Pushable modules perform intermediate transformations on messages flowing between the *Stream head* and *driver*. A *driver* is a non-pushable module and a *Stream head* includes a non-pushable module.
Put procedure	The routine in a QUEUE which receives messages from the preceding QUEUE. It is the single entry point into a QUEUE from a preceding QUEUE. The procedure 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 STREAMS defined set of C-language structures. A module is composed of a read (*upstream*) QUEUE and a write (*downstream*) QUEUE. A QUEUE will typically contain a put and service procedure, a *message queue*, and private data. The read QUEUE (cf. *read queue*) in a *module* will also contain the open procedure and close procedure for the *module*. The primary structure is the **queue_t** structure, occasionally used as a synonym for a QUEUE.
Read queue	The *message queue* in a *module* or *driver* containing *messages* moving *upstream*. Associated with a **read**(2) system call and input from a *driver*.
Schedule	Place a QUEUE on the internal list of QUEUEs which will subsequently have their service procedure called by the STREAMS scheduler.
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	The routine in a QUEUE which receives messages queued for it by the *put procedure* of the QUEUE. The procedure is called by the STREAMS *scheduler*. It may perform processing on the message and will generally pass the message to the *put procedure* of the following QUEUE.
Service provider	In a *service interface*, the entity (typically a *module* or *driver*) that responds to request primitives from the *service user* with response and event primitives.
Service user	In a *service interface*, the entity that generates request primitives for the *service provider* and consumes response and event primitives.
Stream	The kernel aggregate created by connecting STREAMS components, resulting from an application of the STREAMS mechanism. The primary components are the *Stream head*, the *driver*, and zero or more pushable *modules* between the *Stream head* and *driver*.
Stream end	The end of the *Stream* furthest from the user process, containing a *driver*.
Stream head	The end of the *Stream* closest to the user process. It provides the interface between the *Stream* and the user process.
STREAMS	A kernel mechanism that supports development of network services and data communication *drivers*. 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.
Upper stream	A Stream terminating above a multiplexer *pseudo-device driver*. The far end of an upper Stream originates at the *Stream head* or another multiplexer driver.
Upstream	The direction from *driver* towards *Stream head*.

Water marks	Limit values used in *flow control.* Each QUEUE has a high-water mark and a low-water mark. The high-water mark value indicates the upper limit related to the number of characters contained on the *message queue* of a QUEUE. When the enqueued characters in a QUEUE reach its high-water mark, STREAMS causes another QUEUE that attempts to send a message to this QUEUE to become *blocked.* When the characters in this QUEUE are reduced to the low-water mark value, the other QUEUE will be unblocked by STREAMS.
Write queue	The *message queue* in a *module* or *driver* containing *messages* moving *downstream;* associated with a **write**(2) system call and output from a user process.

Index

Accessible Symbols and Functions ... D-4
Accessing the Datagram Provider ... 4-9
Adding and Removing Modules ... 5-6
adjmsg - trim bytes in a message ... C-3
Advanced Flow Control ... 13-5
Advanced Input/Output Facilities ... 2-1
allocb - allocate a message block ... C-3
Appendix A: - Kernel Structures ... A-1
Appendix B: - Message Types ... B-1
Appendix C: Utilities ... C-1
Appendix D: - Design Guidelines ... D-1
Appendix E: - Configuring ... E-1
Asynchronous Input/Output ... 2-7
backq - get pointer to the queue behind a given queue ... C-3
bappend Subroutine ... 7-6
bufcall - recover from failure of allocb ... C-4
Buffer Allocation Priority ... C-16
Building a Multiplexer ... 3-4
canput - test for room in a queue ... C-4
Clone Open ... 2-8
Cloning ... 10-1
Close ... 10-13
Closing ... 5-7
Closing the Service ... 4-13
Configuration Examples ... E-3
Configuration Mechanism ... E-2
Configuring STREAMS Modules and Drivers ... E-1
Connecting Lower Streams ... 11-2
Control of Stream Head Processing ... 13-8
copyb - copy a message block ... C-5
copymsg - copy a message ... C-5
Data Structures ... D-3
Datagram Service Interface Example ... 4-7
datamsg - test whether message is a data message ... C-5
Declarations ... 12-3
Definition ... 12-1
Development Facilities xvii
Disconnecting Lower Streams ... 11-4
Dismantling a Multiplexer ... 3-13
Driver and Module Ioctls ... 9-12
Driver Close ... 9-15
Driver Declarations ... 9-4
Driver Flow Control ... 9-3
Driver Flush Handling ... 9-9
Driver Interrupt ... 9-10
Driver Open ... 9-6
Driver Processing Procedures ... 9-8
Driver Programming ... 9-4
dupb - duplicate a message block descriptor ... C-5
dupmsg - duplicate a message ... C-6
enableok - re-allow a queue to be scheduled for service ... C-6
Error and Trace Logging ... D-7
Example ... 8-8, 12-3
Filter Module Declarations ... 7-5
Flow Control ... 8-6
flushq - flush a queue ... C-6
freeb - free a message block ... C-7
freemsg - free all message blocks in a message ... C-7

General Rules ... D-1
getq - get a message from a queue ... C-7
Glossary G-1
Header Files ... D-3
Input/Output Polling ... 2-2
Inserting Modules ... 1-4
insq - put a message at a specific place in a queue ... C-8
Introduction to this Guide i
iocblk ... A-4
linkb - concatenate two messages into one ... C-8
linkblk ... A-5
Loop-Around Driver ... 10-2
Lower QUEUE Write Service Procedure ... 11-16
Lower Read Put Procedure ... 11-19
Message Allocation ... 7-8
Message Format ... 7-1
Message Generation and Reception ... 7-3
Message Interface ... 4-4
Message Queues and Message Priority ... 8-4
Message Structures ... A-4
Message Usage ... 12-1
Module and Driver Control ... 1-7
Module and Driver Environment ... 6-6
Module Declarations ... 6-1
Module Procedures ... 6-4
msgdsize - get the number of data bytes in a message ... C-8
Multiplexer Configurations ... 3-1
Multiplexer Construction Example ... 11-5
Multiplexing Configurations ... 11-1
Multiplexing Driver ... 11-8
noenable - prevent a queue from being scheduled ... C-9
Notational Conventions i
Opening a Stream ... 5-5
Ordinary Messages ... B-2
OTHERQ - get pointer to the mate queue ... C-9
Overview of Drivers ... 9-1
Priority Messages ... B-10
Procedures ... 8-8
pullupmsg - concatenate bytes in a message ... C-9
Put Procedure ... 7-9
putbq - return a message to the beginning of a queue ... C-10
putctl - put a control message ... C-10
putctl1 - put a control message with a one-byte parameter ... C-10
putnext - put a message to the next queue ... C-11
putq - put a message on a queue ... C-11
qenable - enable a queue ... C-12
qreply - send a message on a stream in the reverse direction ... C-12
qsize - find the number of messages on a queue ... C-12
QUEUE Structures ... A-2
queue_t Structure ... 8-1
RD - get pointer to the read queue ... C-12
Read Options ... 13-8
Receiving a Datagram ... 4-15
Recovering From No Buffers ... 13-1
rmvb - remove a message block from a message ... C-13
rmvq - remove a message from a queue ... C-13
Routing Data Through a Multiplexer ... 3-14
Rules for Put and Service Procedures ... D-5

Sending a Datagram ... 4-13
Service Interface Messages ... 4-1
Service Interface Procedure ... 12-5
Service Interfaces ... 4-1
Service Procedures ... 10-10
Service Procedures ... 8-2
Signals ... 13-6
Simple Stream ... 1-1
splstr - set processor level ... C-13
Stream Construction ... 5-3
Stream Head Messages ... 10-10
STREAMS Overview xiv
streamtab ... A-1
strlog - submit messages for logging ... C-14
System Calls ... D-2
System Error Messages ... E-6
testb - check for an available buffer ... C-14
Tunable Parameters ... E-4
unlinkb - remove a message block from the head of a message ... C-14
Upper Write Put Procedure ... 11-12
Utility Descriptions ... C-3
Utility Routine Summary ... C-17
WR - get pointer to the write queue ... C-15
Write Offset ... 13-9
Write Put Procedure ... 10-6

AT&T UNIX System V/386 Library

- UNIX System V/386 Release 3.2 Utilities Release Notes **AT&T**
- UNIX System V/386 Release 3.2 Streams Primer **AT&T**
- UNIX System V/386 Release 3.2 User's Guide **AT&T**
- UNIX System V/386 Release 3.2 Programmer's Reference Manual **AT&T**
- UNIX System V/386 Release 3.2 Streams Programmer's Guide **AT&T**
- UNIX System V/386 Release 3.2 Network Programmer's Guide **AT&T**
- UNIX System V/386 Release 3.2 Programmer's Guide Vol. I **AT&T**
- UNIX System V/386 Release 3.2 Programmer's Guide Vol. II **AT&T**
- UNIX System V/386 Release 3.2 System Administrator's Guide **AT&T**
- UNIX System V/386 Release 3.2 System Administrator's Reference Manual **AT&T**

PRENTICE HALL, Englewood Cliffs, N.J. 07632

ISBN 0-13-944919-1