

STREAMS Programmer's Guide

2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.



© 1994 Sun Microsystems, Inc.
2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A.

All rights reserved. This product and related documentation are protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or related documentation may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Portions of this product may be derived from the UNIX[®] and Berkeley 4.3 BSD systems, licensed from UNIX System Laboratories, Inc., a wholly owned subsidiary of Novell, Inc., and the University of California, respectively. Third-party font software in this product is protected by copyright and licensed from Sun's font suppliers.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the United States Government is subject to the restrictions set forth in DFARS 252.227-7013 (c)(1)(ii) and FAR 52.227-19.

The product described in this manual may be protected by one or more U.S. patents, foreign patents, or pending applications.

TRADEMARKS

Sun, the Sun logo, Sun Microsystems, Sun Microsystems Computer Corporation, SunSoft, the SunSoft logo, Solaris, SunOS, OpenWindows, DeskSet, ONC, ONC+, and NFS are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and certain other countries. UNIX and OPEN LOOK are registered trademarks of UNIX System Laboratories, Inc., a wholly owned subsidiary of Novell, Inc. PostScript and Display PostScript are trademarks of Adobe Systems, Inc. All other product names mentioned herein are the trademarks of their respective owners.

All SPARC trademarks, including the SCD Compliant Logo, are trademarks or registered trademarks of SPARC International, Inc. SPARCstation, SPARCserver, SPARCengine, SPARCstorage, SPARCware, SPARCcenter, SPARCclassic, SPARCcluster, SPARCdesign, SPARC811, SPARCprinter, UltraSPARC, microSPARC, SPARCworks, and SPARCcompiler are licensed exclusively to Sun Microsystems, Inc. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK[®] and Sun[™] Graphical User Interfaces were developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

X Window System is a product of the Massachusetts Institute of Technology.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.



Contents

1. Introduction.....	1
Introduction to This Guide.....	1
Audience.....	1
Organization.....	1
Code Examples.....	3
Conventions Used.....	3
Other Documentation.....	4
2. Overview of STREAMS.....	5
What Is STREAMS?.....	5
Basic Streams Operations.....	9
STREAMS Components.....	12
Queues.....	12
Messages.....	13
Modules.....	16
Drivers.....	18

Multiplexing	18
Benefits of STREAMS	23
Standardized Service Interfaces	23
Manipulating Modules	24
3. STREAMS Mechanism	29
STREAMS Mechanism Overview	29
STREAMS System Calls	30
Stream Construction	30
Opening a STREAMS Device File	33
Creating a STREAMS-based Pipe	34
Adding and Removing Modules	35
Closing the Stream	35
Stream Construction Example	36
4. STREAMS Processing Routines	43
Put and Service Procedures	43
Put Procedure	44
Service Procedure	45
An Asynchronous Protocol Stream Example	46
Read-Side Processing	49
Write-Side Processing	52
Analysis	52
5. Messages	53
Message Overview	53
Message Types	53

Expedited Data.	55
Message Structure	56
Message Linkage	57
Sending/Receiving Messages	59
Control of Stream Head Processing.	64
Message Queues and Message Priority.	66
The queue Structure	70
Message Processing.	75
Service Interfaces	82
Service Interface Benefits	83
Service Interface Library Example.	86
Message Allocation and Freeing	99
Recovering From No Buffers	102
Releasing Callback Requests	105
Extended STREAMS Buffers.	105
esballoc Example	108
6. Polling and Signaling	111
Input/Output Polling	111
Synchronous Input/Output	112
Asynchronous Input/Output.	116
Signals.	118
Stream as a Controlling Terminal.	119
Job Control	119
Allocation and Deallocation.	122

Hungup Streams	122
Hangup Signals	122
Accessing the Controlling Terminal	123
7. Overview of Modules and Drivers	125
Module and Driver Environment	125
Module and Driver Declarations	126
Module and Driver ioctls	130
General ioctl Processing	132
I_STR ioctl Processing	134
Transparent ioctl Processing	135
Transparent ioctl Messages	138
Transparent ioctl Examples	139
I_LIST ioctl	151
Flush Handling	153
Flushing Priority Bands	156
Device Driver Interface and Driver–Kernel Interface	158
STREAMS Interface	159
Configuring the System for STREAMS Drivers and Modules	160
Design Guidelines	160
Data Structures	165
8. Modules	167
Module Overview	167
Module Procedures	167
Filter Module Example	171

Flow Control	175
Design Guidelines	178
9. Drivers	181
Device Drivers	181
Overview of Drivers	181
Driver Classification	182
Writing a Driver	182
STREAMS Drivers	184
STREAMS Driver Configuration	184
STREAMS Entry Points	185
Printer Driver Example	187
Driver Flush Handling	190
Driver Interrupt	192
Driver Close	194
Driver Flow Control	194
Cloning	195
Loop-Around Driver	198
Design Guidelines	209
10. Multiplexing	211
Overview of Multiplexing	211
Building a Multiplexer	212
Dismantling a Multiplexer	220
Routing Data Through a Multiplexer	222
Connecting / Disconnecting Lower Streams	222

Connecting Lower Streams.	223
Disconnecting Lower Streams	224
Multiplexer Construction Example	225
Multiplexing Driver.	228
Upper Write Put Procedure	232
Upper Write Service Procedure	235
Lower Write Service Procedure	236
Lower Read Put Procedure.	237
Persistent Links	239
Design Guidelines	243
11. STREAMS-Based Pipes and FIFOs.	245
Overview of Pipes and FIFOs.	245
Creating and Opening Pipes and FIFOs	246
Accessing Pipes and FIFOs.	247
Flushing Pipes and FIFOs.	250
Named Streams	250
Unique Connections	254
12. STREAMS-Based Terminal Subsystem.	259
Overview of Terminal Subsystem	259
Line-Discipline Module	261
Hardware Emulation Module	268
STREAMS-based Pseudo-Terminal Subsystem	270
Line-Discipline Module	271
Pseudo-TTY Emulation Module - PTEM	272

Remote Mode	276
Packet Mode	276
Pseudo-TTY Drivers - ptm and pts	277
13. Multi-Threaded STREAMS	283
MT STREAMS Overview	283
MT STREAMS Framework	284
STREAMS Framework Integrity	285
Message Ordering	285
Your MT Options	286
MT SAFE modules.	286
MT UNSAFE modules.	287
Preparing to Port	287
Porting to SunOS 5.x	289
MT SAFE Modules.	290
MT STREAMS perimeters.	290
Perimeter options.	292
MT configuration.	293
qprocson()/qprocsoff()	293
qtimeout()/qbufcall()	294
qwriter()	294
qwait()	295
Asynchronous Callbacks.	295
Close Race Conditions	296
Module unloading and esballoc.	296

Use of q_next	297
MT SAFE Modules using Explicit Locks.	297
Constraints when using locks	297
Preserving message ordering	298
MT UNSAFE Modules	298
Modifying UNSAFE Drivers	299
Caveats	300
New facilities	301
Old Facilities.	301
spl	302
sleep/wakeup	302
Sample Multi-threaded Device Driver	303
Sample Multi-threaded Module with Outer perimeter.	312
A. STREAMS Data Structures	321
streamtab.	321
QUEUE Structures.	322
queue	322
qinit	323
module_info	323
qband.	324
Message Structures	324
iocblk.	325
copyreq	325
copyresp	326

Other Structures	326
striocctl	326
linkblk	327
stroptions	327
B. Message Types	329
Introduction	329
Ordinary Messages	329
M_BREAK	329
M_CTL	330
M_DATA	330
M_DELAY	330
M_IOCTL	331
M_PASSFP	334
M_PROTO	334
M_RSE	335
M_SETOPTS	336
High-Priority Messages	340
M_COPYIN	340
M_COPYOUT	341
M_ERROR	341
M_FLUSH	342
M_HANGUP	343
M_IOCACK	344
M_IOCDATA	344

M_IOCNAK	345
M_PCPROTO	346
M_PCRSE	346
M_PCSIG	346
M_READ	346
M_START and M_STOP	347
M_STARTI and M_STOPI	347
M_UNHANGUP	347
C. STREAMS Utilities.....	349
Introduction	349
Utility Descriptions	350
adjmsg – trim bytes in a message	350
allocb – allocate a message and data block	351
backq – get pointer to the queue behind a given queue ..	351
bcanput – test for flow control in the given priority band .	351
bcanputnext- test for flow control in the given priority band	351
bufcall – recover from failure of allocb	352
canput – test for room in a queue	352
canputnext - test for room in the next queue	353
copyb – copy a message block	353
copymsg – copy a message	353
datamsg – test whether message is a data message	354
dupb – duplicate a message block descriptor	354
dupmsg – duplicate a message	354

enableok – re-allow a queue to be scheduled for service . .	355
esballoc – allocate message and data blocks	355
esbbcall - call function when buffer is available	355
flushband – flush the messages in a given priority band .	356
flushq – flush a queue	356
freeb – free a single message block	356
freemsg – free all message blocks in a message	357
freezestr - freeze a stream	357
getq – get a message from a queue	357
insq – put a message at a specific place in a queue	358
linkb – concatenate two messages into one	358
msgdsize – get the number of data bytes in a message . . .	359
msgpullup - concatenate bytes in a message	359
noenable – prevent a queue from being scheduled	359
OTHERQ – get pointer to the mate queue	359
pullupmsg – concatenate and align bytes in a message . . .	359
put - call a STREAMS put procedure	360
putbq – return a message to the beginning of a queue . . .	360
putctl – put a control message	360
putctl1 – put a control message with a one-byte parameter	361
putnext – put a message to the next queue	361
putnextctl - put a control message.	361
putnextctl1 - put a control message.	361
putq – put a message on a queue	362

qenable – enable a queue	363
qsize – find the number of messages on a queue	364
RD – get pointer to the read queue	364
rmvb – remove a message block from a message	364
rmvq – remove a message from a queue	364
strlog – submit messages for logging	365
testb – check for an available buffer	366
unbufcall – cancel a bufcall request	366
unfreezestr- unfreeze a stream.....	367
WR – get pointer to the write queue	367
DKI Interface	367
New MT perimeter utility routines	367
qbufcall – recover from failure of allocb	368
qunbufcall – cancel a qbufcall request	369
qntimeout – cancel a qtimeout request	369
qwait/qwait_sig –STREAMS perimeter wait routines	369
qwriter - asynchronous STREAMS perimeter upgrade ...	370
Utility Routine Summary	371
D. Debugging	375
Overview of Debugging Facilities	375
Kernel Debug Printing	376
Console Messages	376
STREAMS Error Logging	377
Error and Trace Logging.....	377

Kernel Examination Tools	379
The crash(1M) Command	379
The adb(1) Command	380
The kadb(1M) Command	380
E. Configuration	381
Introduction	381
Configuring STREAMS Drivers and Modules	381
Data Structure Layout	382
modlinkage	382
modldrv	383
modlstrmod	383
dev_ops	383
cb_ops	384
streamtab	385
qinit	386
Entry Points	386
pts example	386
STREAMS Module Configuration	392
Compilation	393
Kernel Loading	393
Checking module type	393
Tunable Parameters	393
Autopush Facility	394
User Interface	395

F. Manual Pages.	399
DDI/DKI Entry Points	400
DDI/DKI Functions.	401
DDI/DKI Data Structures.	403
Glossary	405
Index.	I- 413

Figures



Figure 2-1	Simple Stream	7
Figure 2-2	STREAMS-based Pipe	8
Figure 2-3	Stream to Communications Driver	11
Figure 2-4	Queue Pair Allocation	12
Figure 2-5	A Message	14
Figure 2-6	Messages on a Message Queue	15
Figure 2-7	A Stream in More Detail	17
Figure 2-8	Many-to-one Multiplexer	19
Figure 2-9	One-to-many Multiplexer	19
Figure 2-10	Many-to-many Multiplexer	20
Figure 2-11	Internet Multiplexing Stream	21
Figure 2-12	X.25 Multiplexing Stream	22
Figure 2-13	Protocol Module Portability	25
Figure 2-14	Protocol Migration	26
Figure 2-15	Module Reusability	27
Figure 3-1	Upstream and Downstream Stream Construction	31



Figure 3-2	Stream Queue Relationship	32
Figure 3-3	Case Converter Module	38
Figure 4-1	Idle Stream Configuration Example	48
Figure 4-2	Operational Stream for Example.	49
Figure 4-3	Module Put and Service Procedures.	50
Figure 5-1	Message Form and Linkage.	58
Figure 5-2	Message Ordering in a Queue.	66
Figure 5-3	Message Ordering with One Priority Band.	67
Figure 5-4	Data Structure Linkage.	74
Figure 5-5	Flow Control.	77
Figure 5-6	Protocol Substitution.	84
Figure 5-7	Service Interface.	85
Figure 5-8	Receiving Data	93
Figure 7-1	Flushing The Write-Side of A Stream.	153
Figure 7-2	Flushing The Read-Side of A Stream	154
Figure 7-3	Interfaces Affecting Drivers.	157
Figure 9-1	Device Driver Streams	186
Figure 9-2	Loop-Around Streams	199
Figure 10-1	MultiplexerProtocol Multiplexer.	213
Figure 10-2	Before Link	214
Figure 10-3	IP Multiplexer After First Link	216
Figure 10-4	IP Multiplexer	217
Figure 10-5	TP Multiplexer	219
Figure 10-6	Internet Multiplexer Before Connecting	226
Figure 10-7	Internet Multiplexer After Connecting.	227

Figure 10-8	open() of MUXdriver and Driver1	240
Figure 10-9	Multiplexer After I_PLINK	241
Figure 10-10	Other Users Opening a MUXdriver	242
Figure 11-1	Pushing Modules on a STREAMS-based Pipe	247
Figure 11-2	Server Sets Up a Pipe	255
Figure 11-3	Processes X and Y Open /usr/toserv.	256
Figure 12-1	STREAMS-based Terminal Subsystem.	260
Figure 12-2	Pseudo-TTY Subsystem Architecture.	272
Figure 13-1	Inner perimeter spanning a pair of queues. (D_MPTQPAIR). .	290
Figure 13-2	Inner perimeter spanning all queues in a module. (D_MTPERMOD)	291
Figure 13-3	Outer perimeter spanning all queues in a module with inner perimeters spanning each pair of queues. (D_MTOUTPERIM combined with D_MTQPAIR).	292
Figure B-1	M_PROTO and M_PCPROTO Message Structure	335
Figure D-1	Error and Trace Logging	378



Tables



Table 6-1	Data in siginfo_t structure	118
Table 7-1	Stream open flag	129
Table C-1	Summary of Utility Routines.....	371



Introduction

1 

Introduction to This Guide

This guide provides information to developers on the use of the STREAMS mechanism at user and kernel levels. STREAMS was developed to augment the character input/output mechanism and to support development of communication services.

STREAMS provides developers with integral functions, a set of utility routines, and facilities that speed software design and implementation.

Audience

The guide is intended for network and systems programmers who use the STREAMS mechanism at user and kernel levels for UNIX system communication services.

Readers of the guide are expected to possess prior knowledge of the UNIX system, programming, networking, and data communication. It is also assumed that the reader is familiar with the book *Writing Device Drivers*.

Organization

This guide has several chapters, each discussing a unique topic. Chapters 2, 3, and 4 have introductory information and can be ignored by those readers already familiar with STREAMS concepts and facilities.

- Chapter 1, "Introduction" describes the organization and purpose of the guide. It also defines an intended audience and an expected background of the users of the guide.
- Chapter 2, "Overview of STREAMS" presents an overview and the benefits of STREAMS.
- Chapter 3, "STREAMS Mechanism" describes the basic operations for constructing, using, and dismantling Streams. These operations are performed using `open(2)`, `close(2)`, `read(2)`, `write(2)`, and `ioctl(2)`.
- Chapter 4, "STREAMS Processing Routines" gives an overview of the STREAMS `put` and `service` routines.
- Chapter 5, "Messages" discusses STREAMS messages, their structure, linkage, queuing, and interfacing with other STREAMS components.
- Chapter 6, "Polling and Signaling" describes how STREAMS allows user processes to monitor, control, and poll Streams to allow an effective use of system resources.
- Chapter 7, "Overview of Modules and Drivers" describes the STREAMS module and driver environment, `ioctls`, routines, declarations, flush handling, and driver-kernel interface. It also provides general design guidelines for modules and drivers.
- Chapter 8, "Modules" provides information on module construction and function.
- Chapter 9, "Drivers" discusses STREAMS drivers, elements of driver flow control, flush handling, cloning, and processing.
- Chapter 10, "Multiplexing" describes the STREAMS multiplexing facility.
- Chapter 11, "STREAMS-Based Pipes and FIFOs" provides information on creating MS-based pipes and FIFOs and unique connections.
- Chapter 12, "STREAMS-Based Terminal Subsystem" discusses STREAMS-based terminal and pseudo-terminal subsystems.
- Chapter 13, "Multi-Threaded STREAMS" describes the multithreaded environment and what needs to be done to make your module or driver MT- safe.
- Appendix A, "STREAMS Data Structures" summarizes data structures commonly used by STREAMS modules and drivers.
- Appendix B, "Message Types" describes STREAMS messages and their use.

- Appendix C, “STREAMS Utilities” describes STREAMS utility routines and their use.
- Appendix D, “Debugging” provides debugging aids for developers.
- Appendix E, “Configuration” describes how modules and drivers are configured into the UNIX system, tunable parameters, and the autopush facility.
- Appendix F, “Manual Pages” lists STREAMS-related manual pages.
- “Glossary” defines terms unique to STREAMS.

Code Examples

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

Conventions Used

Throughout this guide, the word “STREAMS” refers to the mechanism and the word “Stream” refers to the path between a user application and a driver. In connection with STREAMS-based pipes, “Stream” refers to the data transfer path in the kernel between the kernel and one or more user processes.

Examples are given to highlight the most important and common capabilities of STREAMS. They are not exhaustive and, for simplicity, reference fictional drivers and modules. Where possible, examples will be runnable code.

Command names, C code, UNIX code, system calls, STREAMS utility routines, header files, data structures, declarations, short examples, filenames, and path names are printed in `listing (constant width) font`.

User input is in `listing font` when by itself, or **bold listing font** when used in combination with computer output.

Screens are used to simulate what a user will see on a video display screen or to show program source code.

Data structure contents and formats are also shown in screens.

Items being emphasized, variable names, and parameters are printed in *italics*.



Warning – The warning sign is used to show possible damage to data, system, application, or person.



Caution – The caution sign is used to show possible harm or damage to a system, an application, a process, or a piece of hardware.

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

Other Documentation

Though the STREAMS Programmer's Guide is a principal tool to aid in developing STREAMS applications, readers are encouraged to obtain more information on both system calls and utilities used by STREAMS from the manual pages. See Appendix F, "Manual Pages" for a complete list of the available manual pages. The actual manual pages are in the SunOS *man Pages(2): System Calls* and the *man Pages(3): Library Routines*.

For more information on driver related issues, such as autoconfiguration, see *Writing Device Drivers*.

For a complete list of books about SunOS/SVR4, see the Solaris Roadmap to Documentation.

STREAMS is also described to some extent in the *System V Interface Definition*.

Overview of STREAMS

2 

What Is STREAMS?

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 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 high performance network services and their components. The *STREAMS* framework itself does not impose any specific network architecture. The *STREAMS* user interface is upwardly compatible with the character I/O user-level functions such as `open()`, `close()`, `read()`, `write()`, and `ioctl()`. Benefits of *STREAMS* are discussed in more detail later in this chapter.

A *Stream* is a full-duplex bidirectional processing and data-transfer path between a *STREAMS* driver in kernel space and a process in user space (see Figure 2-1). 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* nearest the user process. All system calls made by a user-level process on a *Stream* are processed by the *Stream head*.

Pipes are also STREAMS-based. A STREAMS-based pipe (see Figure 2-2) is a full-duplex data transfer path in the kernel. It implements a connection between the kernel and one or more user processes and also shares properties of STREAMS-based devices.

A STREAMS *driver* is 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. The driver transfers data between the kernel and the device and does little or no processing of data other than conversion between data structures used by the STREAMS mechanism and data structures that the device understands.

A STREAMS *module* represents processing functions to be performed on data flowing through the Stream. The module is a defined set of kernel-level routines and data structures used to process data, status, and control information. Data processing may involve changing the way the data is represented, adding/deleting header and trailer information to data, and/or packetizing/depacketizing data. Status and control information includes signals and input/output control information. Each module is self-contained and functionally isolated from any other component in the Stream except its two neighboring components. The module communicates with its neighbors by passing messages. The module is not a required component in STREAMS, whereas the driver is, with the exception of a STREAMS-based pipe where only the Stream head is required.

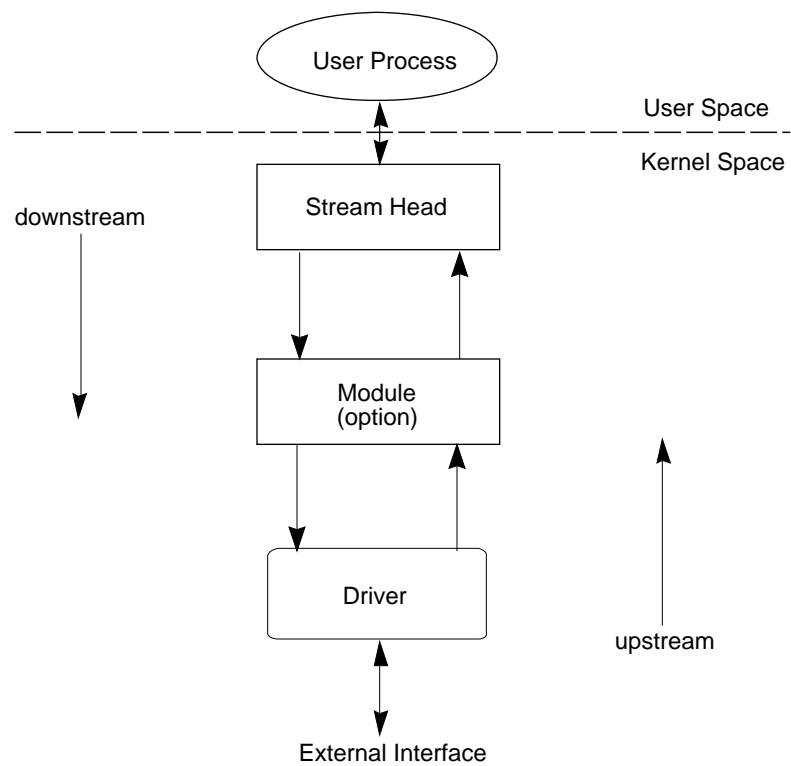


Figure 2-1 Simple Stream

One or more modules may be inserted into a Stream between the Stream head and driver to perform intermediate processing of messages as they pass between the Stream head and driver. STREAMS modules are dynamically connected in a Stream by a user process. No kernel programming, or assembly, is required to create the connection.

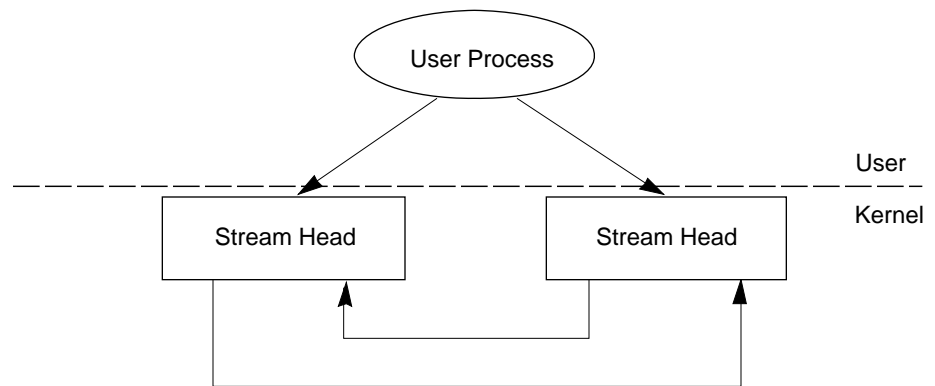


Figure 2-2 STREAMS-based Pipe

STREAMS uses queue structures to keep information about given instances of a pushed module or opened STREAMS device. A *queue* is a data structure that contains status information, a pointer to procedures for processing messages, and pointers for administering the Stream. Queues are always allocated in pairs, one queue for the read side and other for the write side. There is one queue pair for each driver and module and one for the Stream head. The pair of queues is allocated whenever the Stream is opened or the module is pushed (added) onto the Stream.

Data is passed between a driver and the Stream head and between modules in the form of messages. A *message* is a set of data structures used to pass data, status, and control information between user processes, modules, and drivers. Messages passed from the Stream head toward the driver or from the process to the device, are said to travel *downstream* (also called *the write side*). Similarly, messages passed in the other direction, from the device to the process or from the driver to the Stream head, travel *upstream* (also called *the read side*).

A STREAMS message is made up of one or more message blocks. Each *block* is a 3-tuple consisting of a header, a data block, and a data buffer. 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 is 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.

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.

Basic Streams Operations

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

A STREAMS driver is similar to a traditional 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. Each file system entry corresponds to a separate minor device for that driver. Opening different minor devices of a driver causes 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. This is not true with the clone devices.

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 traditional character I/O mechanism.

The `close()` system call closes a device and dismantles the associated Stream when the last open reference to the Stream is given up.

The following code example shows how a simple Stream is used. In the example, the user program interacts with a communications device that provides point-to-point data transfer between two computers. Data written to the device is transmitted over the communications line, and data arriving on the line can be retrieved by reading from the device.

Code Example 2-1 Simple Stream

```
#include <sys/fcntl.h>
#include <stdio.h>
main()
{
    char buf[1024];
    int fd, count;

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

In the example, `/dev/ttya` 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 2-3 shows the state of the Stream following the call to `open()`.

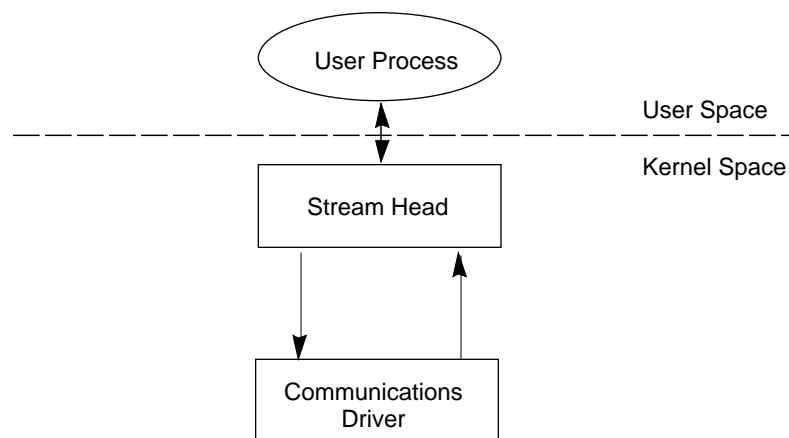


Figure 2-3 Stream to Communications Driver

This example illustrates a user reading data from the communications device and then writing the input back 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 arrive.

Note – The application program must loop on a `read()` call until the desired number of bytes are read. The responsibility for the application getting all the bytes it needs is up to the application programmer, not the STREAMS facilities.

Similarly, the `write()` call attempts to send `count` bytes to `/dev/ttya`. However, STREAMS implements a flow-control mechanism that prevents a user from exhausting system resources by flooding a device driver with data.

Flow control is a STREAMS mechanism that controls the rate of message transfer among the modules, drivers, Stream head, and processes. Flow control is local to each Stream and advisory (voluntary). It limits the number of characters that can be queued for processing at any queue in a Stream. This

mechanism limits buffers and related processing at any queue and in any one Stream, but does not consider buffer pool levels or buffer usage in other Streams. Flow control is not applied to high priority messages. Message priority is discussed in the section “Message Queueing Priority”.

If the Stream exerts flow control, the `write()` call blocks until flow control has been relieved, unless `O_NDELAY` or corresponding POSIX `O_NONBLOCK` flag has been set. The call will not return until it has sent `count` bytes to the device. `exit()` is called to terminate the user process. This system call also closes all open files, dismantling the Stream, and flushes the data.

STREAMS Components

This section gives an overview of the STREAMS components and discusses how these components interact with each other. A more detailed description of each STREAMS component is given in later chapters.

Queues

A queue is an interface between a STREAMS driver or module and the rest of the Stream. Each instance of an open driver or pushed module has a pair of queues allocated, one for read-side and one for write-side. Queues are always allocated as an adjacent pair, similar to an array of structures (see Figure 2-4). The queue with the lower address in the pair is a read queue, and the queue with the higher address is used for the write queue. The `RD()`, `WR()`, and `OTHERQ()` routines move you from one to the other. See man pages `RD(9F)`, `WR(9F)`, `OTHERQ(9F)`, `queue(9S)`.

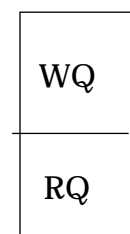


Figure 2-4 Queue Pair Allocation

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

A queue's `put` procedure is invoked by the preceding queue's `put` and/or `service` procedure to add a message to the current queue. If a module does not need to queue messages, its `put` procedure can call the neighboring queue's `put` procedure. Chapter 4, "STREAMS Processing Routines" discusses the `service` and `put` procedures in more detail.

Each queue also has a pointer to an `open` and `close` routine. The `open` routine of a driver is called when the driver is first opened and on every successive open of the Stream. The `open` routine of a module is called when the module is first pushed on the Stream and on every successive open of the Stream. The `close` routine of the module is called when the module is popped (removed) off the Stream. The `close` routine of the driver is called when the last reference to the Stream is given up and the Stream is dismantled.

Messages

All input and output under STREAMS is based on messages. The objects passed between STREAMS modules are pointers to messages. All STREAMS messages use two data structures (`msgb(9S)` and `datab(9S)`) to refer to the message data. These data structures describe the type of the message and contain pointers to the data of the message, as well as other information. Messages are sent through a Stream by successive calls to the `put` procedure of each module or driver in the Stream.

Message Types

All STREAMS messages are assigned message types to indicate their intended use by modules and drivers and to determine their handling by the Stream head. A driver or module can assign most types to a message it generates, and a module can modify a message type during processing. The Stream head will convert certain system calls to specified message types and send them downstream, and it will respond to other calls by copying the contents of certain message types that were sent upstream.

Most message types are internal to STREAMS and can only be passed from one STREAMS component to another. A few message types, for example `M_DATA`, `M_PROTO`, and `M_PCPROTO`, can also be passed between a Stream and user processes. `M_DATA` messages carry data within a Stream and between a Stream and a user process. `M_PROTO` or `M_PCPROTO` messages carry both data and control information.

As shown in Figure 2-5, a STREAMS message consists of one or more linked message blocks that are attached to the first message block of the same message.

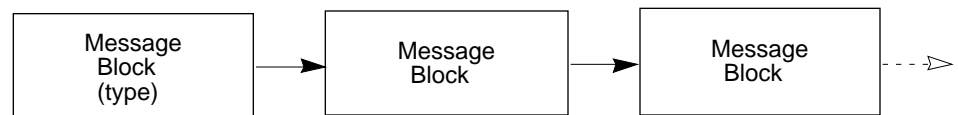


Figure 2-5 A Message

Messages can exist stand-alone, as in Figure 2-5, when the message is being processed by a procedure. Alternately, a message can await processing on a linked list of messages, called a message queue. In Figure 2-6, Message 2 is linked to Message 1.

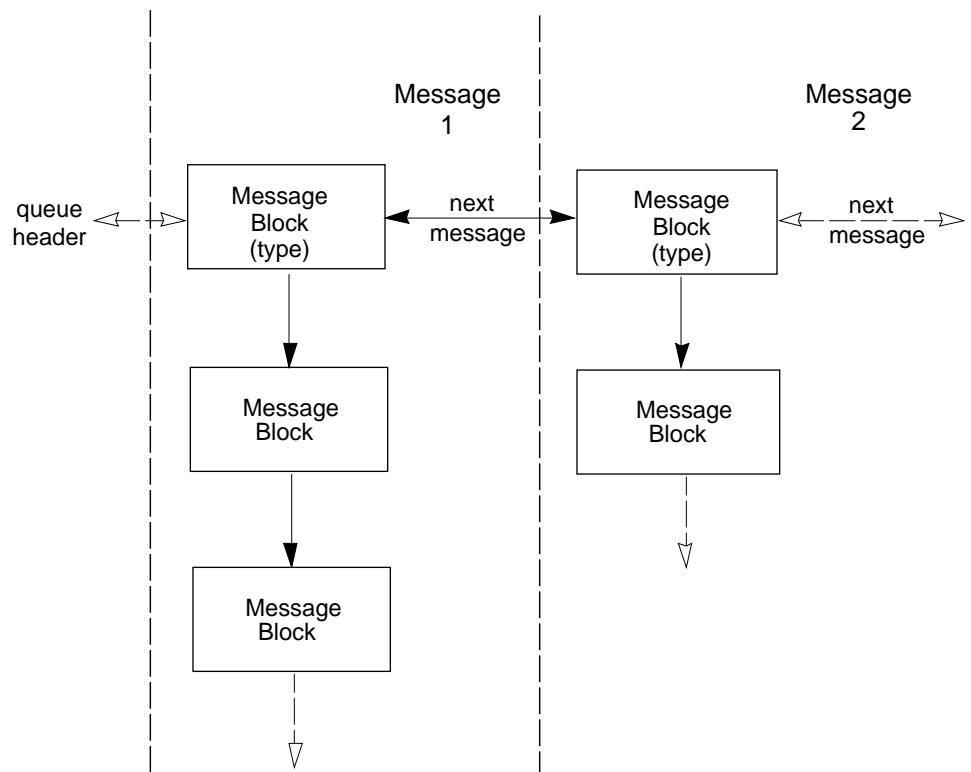


Figure 2-6 Messages on a Message Queue

When a message is in a queue, the first block of the message contains links to preceding and succeeding messages on the same message queue, in addition to the link to the second block of the message (if present). The message queue head and tail are contained in the queue.

STREAMS utility routines lets developers manipulate messages and message queues.

Message Queueing Priority

In certain cases, messages containing urgent information (such as a break or alarm conditions) must pass through the Stream quickly. To accommodate these cases, STREAMS provides multiple classes of message queueing priority.

All messages have an associated priority field. Normal (ordinary) messages have a priority of zero. Priority messages have a priority greater than zero. High-priority messages are high priority by virtue of their message type. By convention, STREAMS prevents high-priority messages from being blocked by flow control and causes a `service` procedure to process them ahead of all ordinary messages on the queue. This results in the high-priority message transiting each module with minimal delay.

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

Message priority is defined by the message type. High-priority message types cannot be changed to be normal message types. Certain message types come in equivalent high priority/ordinary pairs (for example, `M_PCPROTO` and `M_PROTO`), so that a module or device driver can choose between the two priorities when sending information.

Modules

A module performs intermediate transformations on messages passing between a Stream head and a driver. There may be zero or more modules in a Stream (zero when the driver performs all the required character and device processing).

Each module is constructed from a pair of queue structures (see “Au/Ad” and “Bu/Bd” in Figure 2-7). One queue performs functions on messages passing upstream through the module (“Au” and “Bu” in Figure 2-7). The other set (“Ad” and “Bd”) performs another set of functions on downstream messages.

Each of the two queues in a module will generally have distinct functions, that is, unrelated processing procedures and data. The queues operate independently and “Au” will not know if a message passes through “Ad” unless “Ad” is programmed to inform it. Messages and data can be shared only if the developer specifically programs the module functions to perform the sharing.

Each queue can directly access the adjacent queue in the direction of message flow (for example, “Au” to “Bu” or “Bd” to “Ad”). In addition, within a module, a queue can readily locate its mate and access its messages and data.

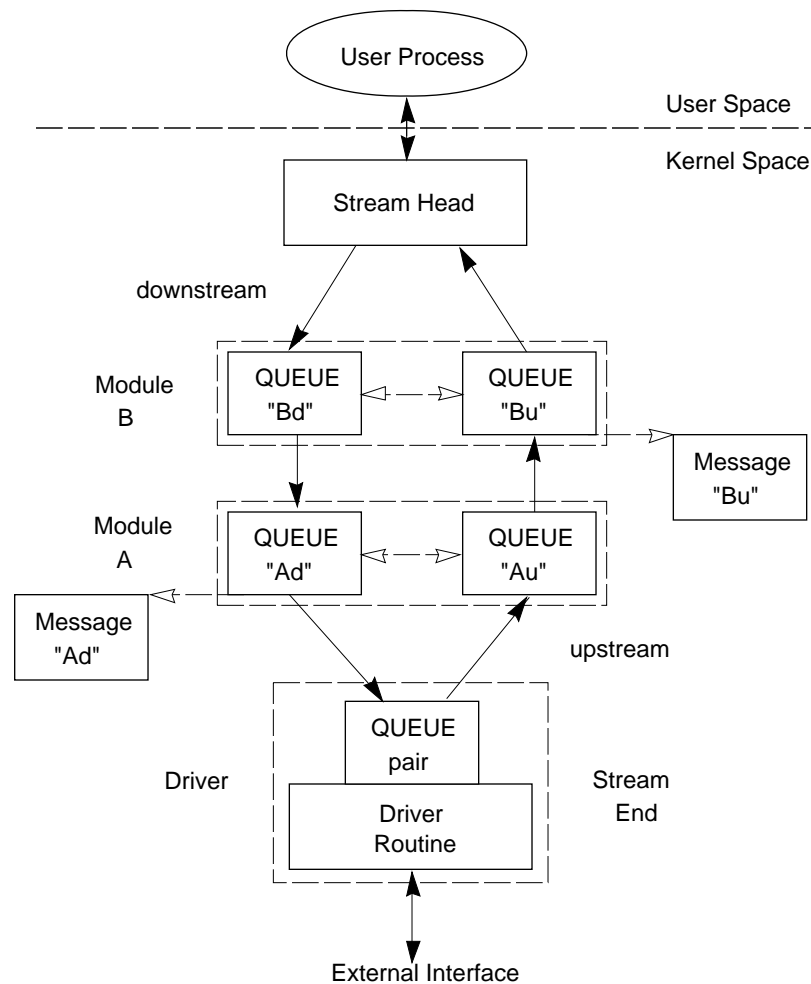


Figure 2-7 A Stream in More Detail

Each queue in a module points to messages, processing procedures, and data:

- Messages – These are dynamically attached to the queue on a linked list (“message queue”, see “Ad” and “Bu” in Figure 2-7) as they pass through the module.
- Processing procedures – A `put` procedure processes messages and must be incorporated in each queue. An optional `service` procedure can also be incorporated. According to their function, the procedures can send messages upstream and/or downstream, and they can also modify the private data in their module.
- Data – You may use a private field in the queue to reference private data structures (for example, state information and translation tables).

In general, each of the two queues in a module has a distinct set of all of these elements.

Drivers

STREAMS device drivers are an initial part of a Stream. They are structurally similar to STREAMS modules. The call interfaces to driver routines are identical to the interfaces used for modules.

There are three significant differences between modules and drivers. A driver must be able to handle interrupts from the device, a driver can have multiple Streams connected to it, and a driver is initialized/deinitialized via `open` and `close`. A module may be initialized by either an `I_PUSH` `ioctl` (and thus deinitialized via the `I_POP` `ioctl`) or an `open`. Modules are pushed automatically during an `open` if a stream has been configured by the `autopush(1M)` mechanism.

Drivers and modules can pass signals, error codes, and return values to processes via message types provided for that purpose.

Multiplexing

Earlier, Streams were described as linear connections, or chains 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 2-8. 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.

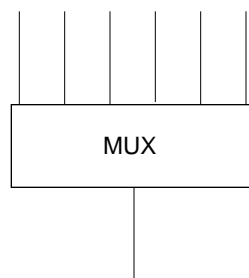


Figure 2-8 Many-to-one Multiplexer

A second type of multiplexer might route data from a single upper Stream to one of several lower Streams, as shown in Figure 2-9. An internetworking protocol could take this form, where each lower Stream links the protocol to a different physical network.

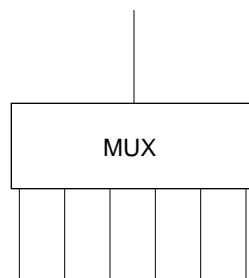


Figure 2-9 One-to-many Multiplexer

A third type of multiplexer might route data from one of many upper Streams to one of many lower Streams, as shown in Figure 2-10.

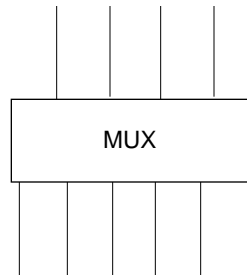


Figure 2-10 Many-to-many Multiplexer

The STREAMS mechanism supports the multiplexing of Streams through special pseudo-device drivers. Using a linking facility mechanism within the STREAMS framework, users can dynamically build, maintain, and dismantle multiplexed Stream configurations. Simple configurations like the ones shown in three previous figures can be further combined to form complex, multilevel multiplexed Stream configurations.

STREAMS multiplexing configurations are created in the kernel by interconnecting multiple Streams. Conceptually, there are two kinds of multiplexers: upper and lower multiplexers. *Lower multiplexers* have multiple lower Streams between device drivers and the multiplexer, and *upper multiplexers* have multiple upper Streams between user processes and the multiplexer.

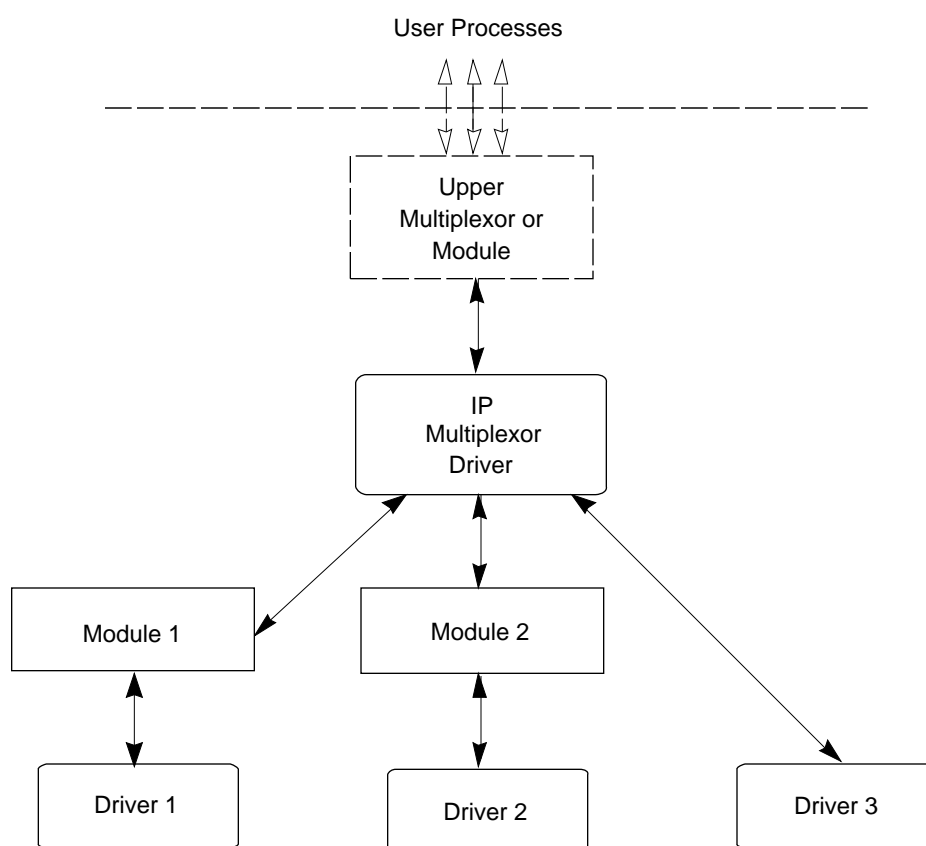


Figure 2-11 Internet Multiplexing Stream

Figure 2-11 is an example of the multiplexer configuration that would typically occur where internetworking functions were included in the system. This configuration contains three hardware device drivers. The IP (Internet Protocol) is a multiplexer.

The IP multiplexer switches messages among the lower Streams or sends them upstream to user processes in the system. In this example, the multiplexer expects to see the same interface downstream to Module 1, Module 2, and Driver 3.

Figure 2-11 shows the IP multiplexer as part of a larger configuration. The multiplexer configuration, as shown in the dashed rectangle, would generally have an upper multiplexer and additional modules. Multiplexers could also be cascaded below the IP multiplexer driver if the device drivers were replaced by multiplexer drivers.

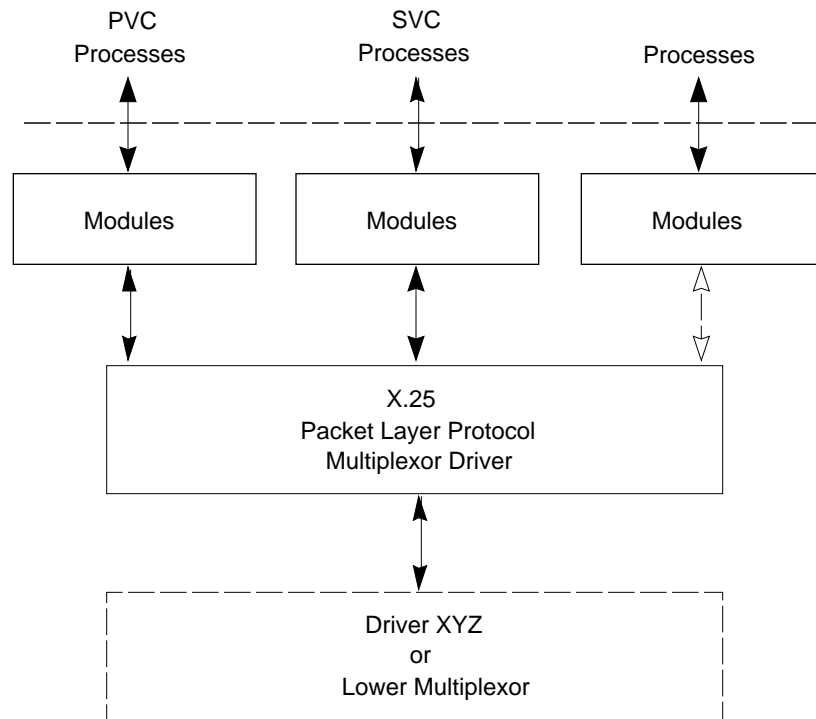


Figure 2-12 X.25 Multiplexing Stream

Figure 2-12 shows a multiplexer configuration where the multiplexer (or multiplexing driver) routes messages between the lower Stream and one of the upper Streams. This Stream performs X.25 multiplexing to multiple independent SVC (Switched Virtual Circuit) and PVC (Permanent Virtual Circuit) user processes. Upper multiplexers are a specific application of standard STREAMS facilities that support multiple minor devices in a device

driver. This figure also shows that more complex configurations can be built by having one or more multiplexed drivers below and multiple modules above an upper multiplexer.

You can choose either upper or lower multiplexing, or both, when designing their applications. For example, a window multiplexer would have a similar configuration to the X.25 configuration of Figure 2-12, with a window driver replacing Packet Layer, a tty driver replacing the driver XYZ, and the child processes of the terminal process replacing the user processes. Although the X.25 and window multiplexing Streams have similar configurations, their multiplexer drivers would differ significantly. The IP multiplexer of Figure 2-11 has a different configuration than the X.25 multiplexer, and the driver would implement its own set of processing and routing requirements in each configuration.

In addition to upper and lower multiplexers, more complex configurations can be created by connecting Streams containing multiplexers to other multiplexer drivers. With such a diversity of needs for multiplexers, it is not possible to provide general purpose multiplexer drivers. Rather, STREAMS provides a general purpose multiplexing facility. The facility allows users to set up the inter-module/driver plumbing to create multiplexer configurations of generally unlimited interconnection.

Benefits of STREAMS

STREAMS provides a flexible, portable, and reusable set of tools for development of UNIX system communication services. STREAMS allows an easy creation of modules that offer standard data communications services and the ability to manipulate those modules on a Stream. From user level, modules can be dynamically selected and interconnected; kernel programming, assembly, and link editing are not required to create the interconnection.

STREAMS also greatly simplifies the user interface for languages that have complex input and output requirements. This is discussed in Chapter 12, "STREAMS-Based Terminal Subsystem".

Standardized Service Interfaces

STREAMS simplifies the creation of modules that present a service interface to any neighboring application program, module, or device driver. A service interface is defined at the boundary between two neighbors. In STREAMS, a

service interface is a set of messages and the rules that allow passage of these messages across the boundary. A module that implements a service interface will receive a message from a neighbor and respond with an appropriate action (for example, send back a request to retransmit) based on the specific message received and the preceding sequence of messages.

In general, any two modules can be connected anywhere in a Stream. However, rational sequences are generally constructed by connecting modules with compatible protocol service interfaces. For example, a module that implements an X.25 protocol layer, as shown in Figure 2-13, presents a protocol service interface at its input and output sides. In this case, other modules should only be connected to the input and output side if they have the compatible X.25 service interface.

Manipulating Modules

STREAMS provides the abilities to manipulate modules from user level, to interchange modules with common service interfaces, and to change the service interface to a STREAMS user process. These capabilities yield further benefits when implementing networking services and protocols, including:

- User level programs can be independent of underlying protocols and physical communication media.
- Network architectures and higher level protocols can be independent of underlying protocols, drivers, and physical communication media.
- Higher level services can be created by selecting and connecting lower level services and protocols.

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

Protocol Portability

Figure 2-13 shows how the same X.25 protocol module can be used with different drivers on different machines by implementing compatible service interfaces. The X.25 protocol module interfaces are Connection Oriented Network Service (CONS) and Link Access Protocol – Balanced (LAPB).

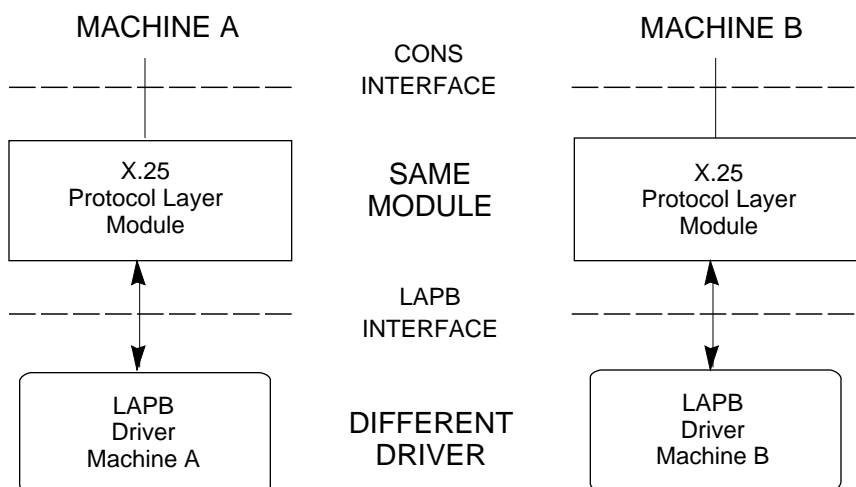


Figure 2-13 Protocol Module Portability

Protocol Substitution

Alternate protocol modules (and device drivers) can be exchanged on the same machine if they are implemented to an equivalent service interface.

Protocol Migration

Figure 2-14 illustrates how STREAMS can move functions between kernel software and front end firmware. A common downstream service interface allows the transport protocol module to be independent of the number or type of modules below. The same transport module will connect without modification to either an X.25 module or X.25 driver that has the same service interface.

By shifting functions between software and firmware, you can produce cost-effective, functionally equivalent systems over a wide range of configurations. They can rapidly incorporate technological advances. The same transport protocol module can be used on a lower capacity machine, where economics may preclude the use of front-end hardware, and also on a larger scale system where a front-end is economically justified.

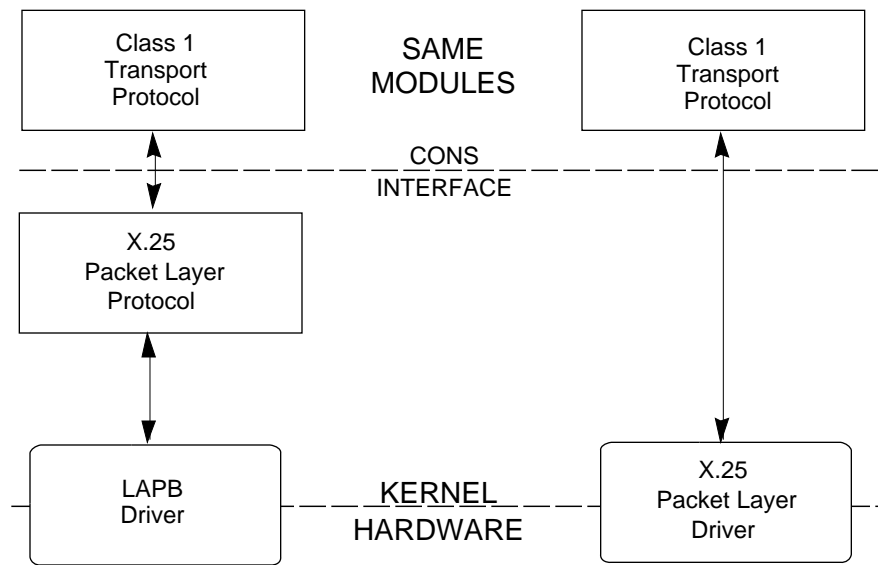


Figure 2-14 Protocol Migration

Module Reusability

Figure 2-15 shows the same canonical module (for example, one that provides delete and kill processing on character strings) reused in two different Streams. This module would typically be implemented as a filter, with no downstream service interface. In both cases, a tty interface is presented to the Stream's user process since the module is nearest the Stream head.

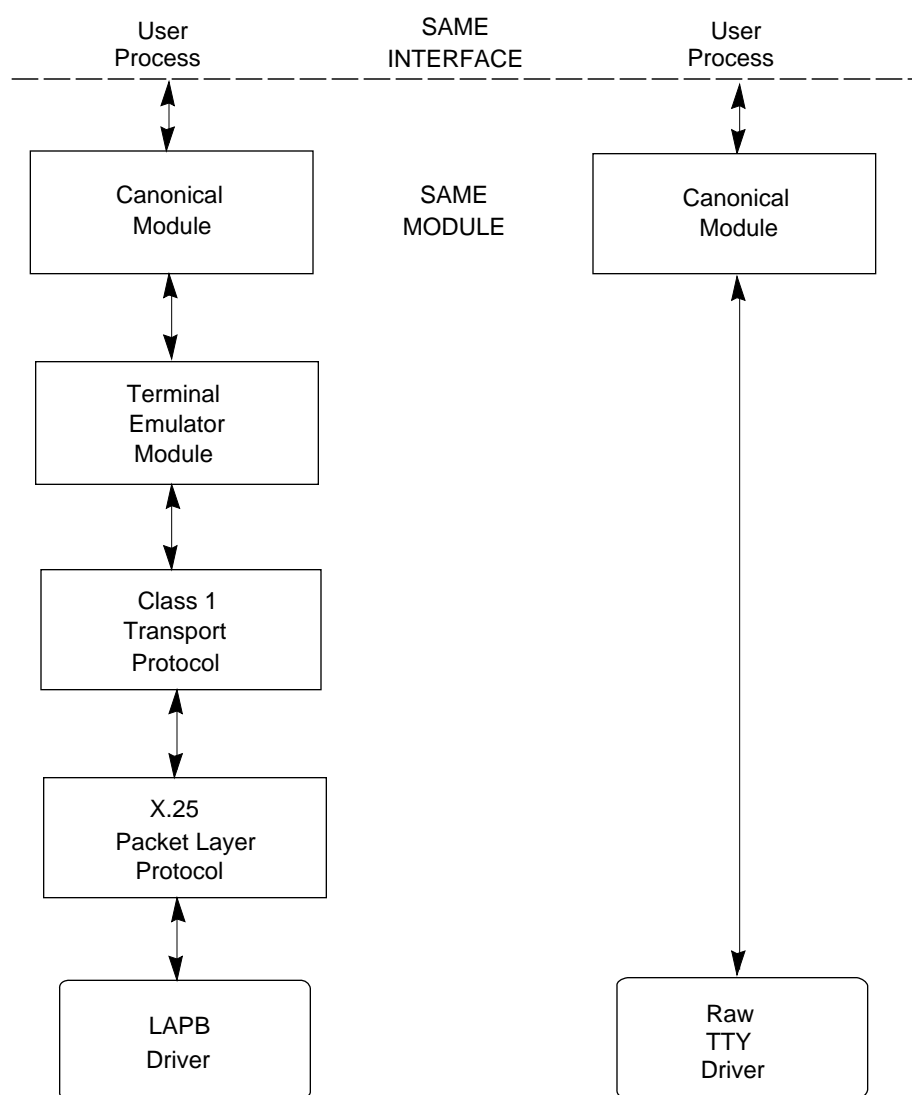


Figure 2-15 Module Reusability

STREAMS Mechanism Overview

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

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

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

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

STREAMS System Calls

The STREAMS-related system calls are:

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

Stream Construction

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

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

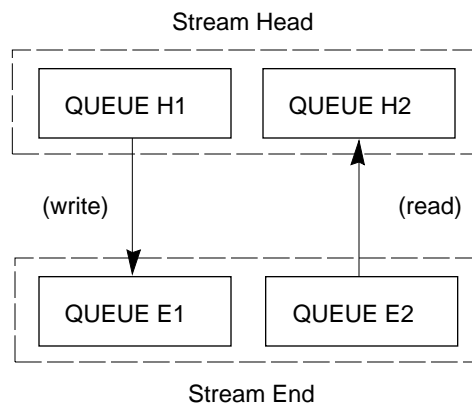


Figure 3-1 Upstream and Downstream Stream Construction

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

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

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

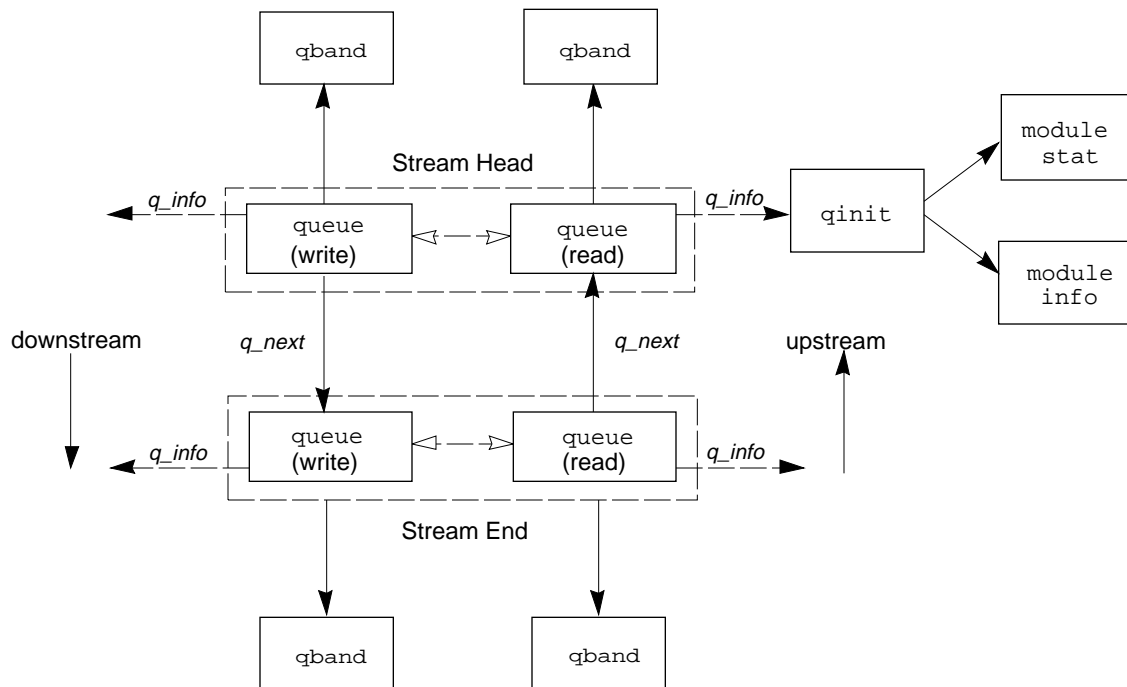


Figure 3-2 Stream Queue Relationship

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

Opening a STREAMS Device File

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

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

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

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

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

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

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

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

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

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

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

Creating a STREAMS-based Pipe

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

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

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

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

Adding and Removing Modules

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

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

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

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

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

Closing the Stream

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

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

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

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

Stream Construction Example

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

Inserting Modules

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

Code Example 3-1 Header Definition

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

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

Code Example 3-2 Pushing a Module

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

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

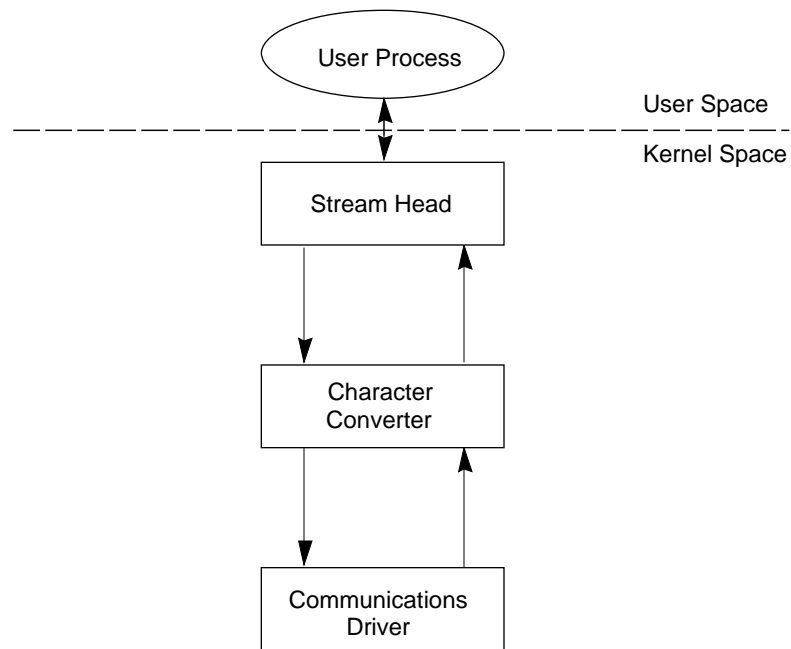


Figure 3-3 Case Converter Module

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

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

Module and Driver Control

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

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

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

Code Example 3-3 Processing Ioctl

```
/* change all uppercase vowels to lowercase */
striocctl.ic_cmd = XCASE;
striocctl.ic_timeout = 0; /* default timeout (15 sec) */
striocctl.ic_dp = "AEIOU";
striocctl.ic_len = strlen(striocctl.ic_dp);
if (ioctl(fd, I_STR, &striocctl) < 0) {
    perror("ioctl I_STR failed");
    exit(3);
}
/* delete all instances of the chars 'x' and 'X' */
striocctl.ic_cmd = DELETE;
striocctl.ic_dp = "xX";
striocctl.ic_len = strlen(striocctl.ic_dp);
if (ioctl(fd, I_STR, &striocctl) < 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 `strioc` structure, which specifies the request to be made to a module or driver. This structure is defined in `streamio(7)` and has the following format:

```
struct strioc {
    int      ic_cmd;           /* ioctl request */
    int      ic_timeout;       /* 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_timeout* specifies the number of seconds an `I_STR` request should wait for an acknowledgment before timing out, *ic_len* is the number of bytes of data to accompany the request, and *ic_dp* points to that data.

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

`I_STR` is intercepted by the Stream head, which packages it into a message, using information contained in the `strioc` structure, and sends the message downstream. Any module that does not understand the command in *ic_cmd* should pass the message further downstream. The request will be processed by the module or driver closest to the Stream head that understands the command specified by *ic_cmd*. The `ioctl` call will block up to *ic_timeout* 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_timeout* seconds, the `ioctl` call will fail.

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

The `strioc1` structure is also used to retrieve the results, if any, of an `I_STR` request. If data is returned by the target module or driver, `ic_dp` must point to a buffer large enough to hold that data, and `ic_len` will be set on return to indicate the amount of data returned.

The remainder of this example is identical to the example in Chapter 2, “Overview of STREAMS”.

Code Example 3-4 Process input

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

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

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

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

STREAMS Processing Routines

4 

Put and Service Procedures

The `put(9E)` and `srv(9E)` procedures in the queue are routines that process messages as they transit the queue. The processing is generally performed according to the message type and can result in a modified message, new message(s), or no message. A resultant message, if any, is generally sent in the same direction in which it was received by the queue, but may be sent in either direction. Each `put(9E)` procedure places messages on its queue as they arrive, for later processing by the `service` procedure.

A queue will always contain a `put` procedure and may also contain an associated `service` procedure. Having both a `put(9E)` and `srv(9E)` procedure in a queue enables STREAMS to provide the rapid response and the queuing required in multi-user systems.

The `service` and `put` procedures pointed at by a queue, and the queues themselves, are not associated with any process. These procedures may not block if they cannot continue processing, but must instead return. Any information about the current status of the queue must be saved by the procedure before returning.

Put Procedure

A `put` procedure is the queue routine that receives messages from the preceding queues in the Stream. Messages are passed between queues by a procedure in one queue calling the `put` procedure contained in the following queue. A call to the `put` procedure in the appropriate direction is generally the only way to pass messages between STREAMS components. There is usually a separate `put` procedure for the read and write queues because of the full-duplex operation of most Streams. However, there can be a single `put` procedure shared between both the read and write queues. The syntax for the `put` procedure is shown in the following example. For more information, see Appendix F, “Manual Pages” for more information.

```
int prefixrput(queue_t *q, mblk_t *mp);
int prefixwput(queue_t *q, mblk_t *mp);
```

where *q* is a pointer to the `queue(9S)` structure, and *mp* is a pointer to the message block. See `msgb(9S)`.

The `put` procedure allows rapid response to certain data and events, such as echoing of input characters. It has higher priority than any scheduled `service` procedure and is associated with immediate, as opposed to deferred, message processing.

The `put` procedure always executes before the `service` routine for each specific message. In the multithreaded environment, a `put` procedure and a scheduled `service` procedure may be running simultaneously in different threads.

Each STREAMS component accesses the adjacent `put` procedure indirectly using the routine `putnext(9F)`.

Note – A module should never directly call other module routines, including `put` and `service` procedures.

For example, consider that *modA*, *modB*, and *modC* are three consecutive components in a Stream, with *modC* connected to the Stream head. If *modA* receives a message to be sent upstream, *modA* processes that message and calls *modB*'s read `put` procedure, which processes it and calls *modC*'s read `put` procedure, which processes it and calls the Stream head's read `put` procedure. Thus, the message will be passed along the Stream in one continuous

processing sequence. This sequence completes the entire processing in a short time with low overhead (subroutine calls). On the other hand, if this sequence is lengthy and the processing is implemented on a multi-user system, then this manner of processing may be good for this Stream but may be detrimental for others. Streams may have to wait too long to get their turn, since each `put` procedure is called from the preceding one, and the kernel stack (or interrupt stack) grows with each function call. The possibility of running off the stack exists, thus panicking the system or producing unpredictable results.

Note – STREAMS modules do not know which modules they are connected to, so the `put` procedures cannot depend on a message being handled only by `put` procedures at the stream head or in the driver. Any other modules along the Stream may queue the message and process it with a `service` procedure.

Service Procedure

In addition to the `put` procedure, a `service` procedure may be contained in each queue to allow deferred message processing. If a queue has both a `put` and `service` procedure, message processing will generally be divided between the procedures. The `put` procedure is always called first, from a preceding queue. After completing its part of the message processing, it arranges for the `service` procedure to be called by passing the message to the `putq(9F)` procedure. `putq` does two things: it places the message on the message queue of the queue and links the queue to the end of the STREAMS scheduling queue. When `putq()` returns to the `put` procedure, the procedure can return or continue to process the message. Some time later, the `service` procedure will be automatically called by the STREAMS scheduler. The syntax for the `service` procedure looks like this:

```
int prefixsrv(queue_t *q);
int prefixwsrv(queue_t *q);
```

The STREAMS scheduler is separate and distinct from the system process scheduler. It is concerned only with queues linked to the STREAMS scheduling queue. The scheduler calls each `service` procedure of the scheduled queues one at a time in a First-In-First-Out (FIFO) manner. There is a dedicated thread for `service` procedure scheduling. `Put` procedures may be called on any thread in the kernel.

Note – The scheduling of queue `service` procedures is machine dependent and implementation specific. However, given the multithreaded architecture, service routines may not have finished running before returning to user level.

STREAMS utilities deliver the messages to the processing `service` procedure in the FIFO manner within each priority class (high priority, priority band, ordinary), because the `service` procedure is unaware of the message priority and simply receives the next message. The `service` procedure receives control in the order it was scheduled. When the `service` procedure receives control, it may encounter multiple messages on its message queue. This buildup can occur if there is a long interval between the time a message is queued by a `put` procedure and the time that the STREAMS scheduler calls the associated `service` procedure. In this interval, there can be multiple calls to the `put` procedure causing multiple messages to build up. The `service` procedure must always process all messages on its message queue unless prevented by flow control.

A `service` procedure can use a `putbq(9F)` to put messages back on a queue due to flow control or other reasons. A high-priority message must *never* be put back on the queue. This may result in an infinite loop in calling the `service` procedure. High priority messages would usually not get queued in the first place.

Terminal output and input erase and kill processing, for example, would typically be performed in a `service` procedure because this type of processing does not have to be as timely as echoing. Use of a `service` procedure also allows processing time to be more evenly spread among multiple Streams. As with the `put` procedure there can be a separate `service` procedure for each queue in a STREAMS component or a single procedure used by both the read and write queues.

Rules that should be observed in `put` and `service` procedures are listed in Chapter 7, “Overview of Modules and Drivers”.

An Asynchronous Protocol Stream Example

In the following example, the computer supports different kinds of asynchronous terminals, each logging in on its own port. The port hardware is limited in function; for example, it detects and reports line and modem status, but does not check parity.

Communications software support for these terminals is provided via a STREAMS based asynchronous protocol. The protocol includes a variety of options that are set when a terminal operator dials in to log on. The options are determined by a STREAMS user process, *getstrm*, which analyzes data sent to it through a series of dialogs (prompts and responses) between the process and terminal operator.

The process sets the terminal options for the duration of the connection by pushing modules onto the Stream or by sending control messages to cause changes in modules (or in the device driver) already on the Stream. The options supported include:

- ASCII or EBCDIC character codes
- For ASCII code, the parity (odd, even or none)
- Echo or not echo input characters
- Canonical input and output processing or transparent (raw) character handling

These options are set with the following modules:

CHARPROC

Provides input character processing functions, including dynamically settable (via control messages passed to the module) character echo and parity checking. The module's default settings are to echo characters and not check character parity.

CANONPROC

Performs canonical processing on ASCII characters upstream and downstream (note that this performs some processing in a different manner from the standard UNIX system character I/O tty subsystem).

ASCEBC

Translates EBCDIC code to ASCII upstream and ASCII to EBCDIC downstream.

At system initialization a user process, *getstrm*, is created for each tty port. *getstrm* opens a Stream to its port and pushes the CHARPROC module onto the Stream by use of an `ioctl I_PUSH` command. Then, the process issues a `getmsg` system call to the Stream and sleeps until a message reaches the Stream head. The Stream is now in its idle state.

The initial idle Stream, shown in Figure 4-1, contains only one pushable module, `CHARPROC`. The device driver is a limited-function raw tty driver connected to a limited-function communication port. The driver and port transparently transmit and receive one unbuffered character at a time.

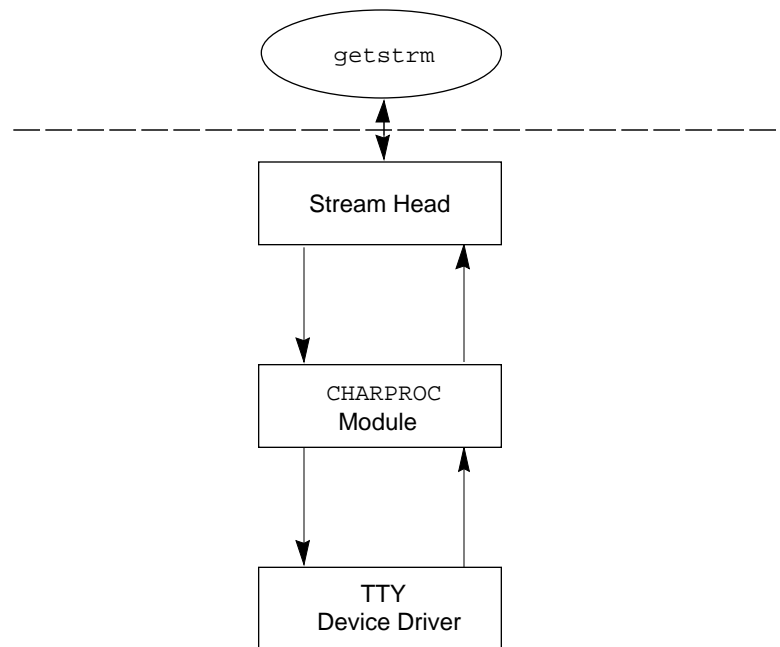


Figure 4-1 Idle Stream Configuration Example

Upon receipt of initial input from a tty port, `getstrm` establishes a connection with the terminal, analyzes the option requests, verifies them, and issues STREAMS system calls to set the options. After setting up the options, `getstrm` creates a user application process. Later, when the user terminates that application, `getstrm` restores the Stream to its idle state by use of similar system calls.

The following figure continues the example and associates kernel operations with user-level system calls. As a result of initializing operations and pushing a module, the Stream for port one has the following configuration:

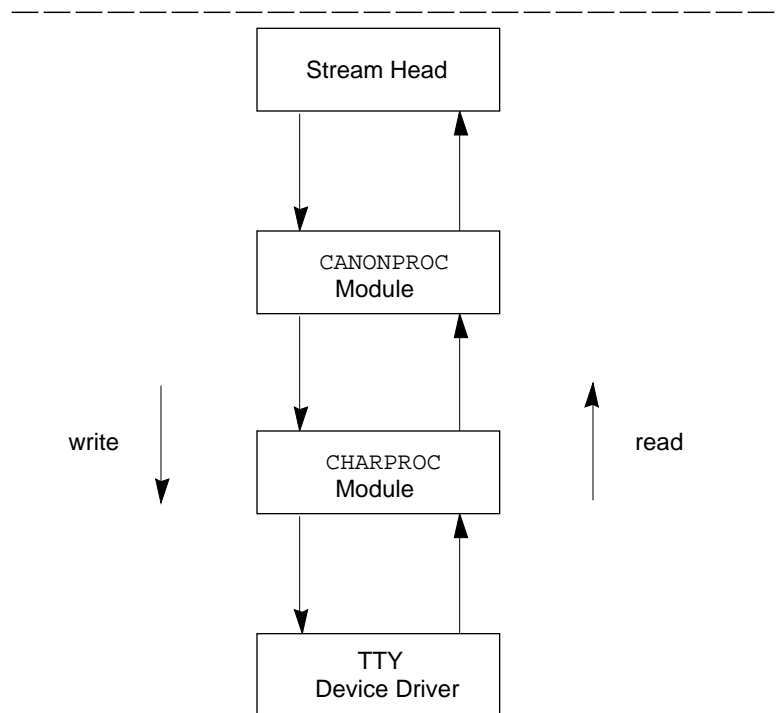


Figure 4-2 Operational Stream for Example

The upstream queue is also referred to as the read queue reflecting the message flow direction. Correspondingly, downstream is referred to as the write queue.

Read-Side Processing

In the example, read-side processing consists of driver processing, CHARPROC processing, and CANONPROC processing.

Driver Processing

The user process has been blocked on the `getmsg(2)` system call while waiting for a message to reach the Stream head, and the device driver independently waits for input of a character from the port hardware or for a message from

upstream. Upon receipt of an input character interrupt from the port, the driver places the associated character in an `M_DATA` message, allocated previously. Then, the driver sends the message to the `CHARPROC` module by calling `CHARPROC`'s upstream `put` procedure. On return from `CHARPROC`, the driver calls the `allocb()` utility routine to get another message for the next character.

CHARPROC

`CHARPROC` has both `put` and `service` procedures on its read-side. In the example, the other queues in the modules also have both procedures:

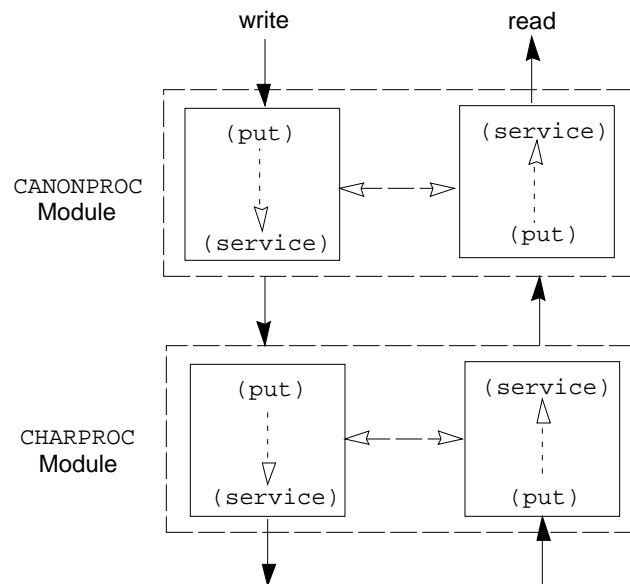


Figure 4-3 Module Put and Service Procedures

When the driver calls `CHARPROC`'s read queue `put` procedure, the procedure checks private data flags in the queue. In this case, the flags indicate that echoing is to be performed (recall that echoing is optional and that you are working with port hardware that cannot automatically echo). `CHARPROC` causes the echo to be transmitted back to the terminal by first making a copy of

the message with a STREAMS utility routine. Then, CHARPROC uses another utility routine to obtain the address of its own write queue. Finally, the CHARPROC read put procedure calls its write put procedure and passes it the message copy. The write procedure sends the message to the driver to effect the echo and then returns to the read procedure.

This part of read-side processing is implemented with put procedures so that the entire processing sequence occurs as an extension of the driver input character interrupt. The CHARPROC read and write put procedures appear as subroutines (nested in the case of the write procedure) to the driver. This manner of processing is intended to produce the character echo quickly.

After returning from echo processing, the CHARPROC read put procedure checks another of its private data flags and determines that parity checking should be performed on the input character. Parity should most reasonably be checked as part of echo processing. However, for this example, parity is checked only when the characters are sent upstream. This relaxes the timing in which the checking must occur, that is, it can be deferred along with the canonical processing. CHARPROC uses putq() to schedule the (original) message for parity check processing by its read service procedure. When the CHARPROC read service procedure is complete, it forwards the message to the read put procedure of CANONPROC. Note that if parity checking was not required, the CHARPROC put procedure would call the CANONPROC put procedure directly.

CANONPROC

CANONPROC performs canonical processing. As implemented, all read queue processing is performed in its service procedure so that CANONPROC's put procedure simply calls putq() to schedule the message for its read service procedure and then exits. The service procedure extracts the character from the message buffer and places it in the *line buffer* contained in another M_DATA message it is constructing. Then, the message that contained the single character is returned to the buffer pool. If the character received was not an end-of-line character, CANONPROC's service procedure returns. Otherwise, a complete line has been assembled and CANONPROC sends the message upstream to the Stream head, which unblocks the user process from the getmsg(2) call and passes it the contents of the message.

Write-Side Processing

The write-side of this Stream carries two kinds of messages from the user process: `M_IOCTL` messages for `CHARPROC`, and `M_DATA` messages to be output to the terminal.

`M_IOCTL` messages are sent downstream as a result of an `ioctl(2)` system call. When `CHARPROC` receives an `M_IOCTL` message type, it processes the message contents to modify internal flags and then uses a utility routine to send an acknowledgment message upstream to the Stream head. The Stream head acts on the acknowledgment message by unblocking the user from the `ioctl`.

For terminal output, it is presumed that `M_DATA` messages, sent by `write(2)` system calls, contain multiple characters. In general, `STREAMS` returns to the user process immediately after processing the `write` call so that the process may send additional messages. Flow control will eventually block the sending process. The messages can queue on the write-side of the driver because of character transmission timing. When a message is received by the driver's write `put` procedure, the procedure will use `putq()` to place the message on its write-side `service` message queue if the driver is currently transmitting a previous message buffer. However, there is generally no write queue `service` procedure in a device driver. Driver output interrupt processing takes the place of scheduling and performs the `service` procedure functions, removing messages from the queue.

Analysis

For efficiency, a module implementation would generally avoid placing one character per message and using separate routines to echo and parity check each character, as was done in this example. Nevertheless, even this design yields potential benefits. Consider a case where alternate, more intelligent, port hardware was substituted. If the hardware processed multiple input characters and performed the echo and parity checking functions of `CHARPROC`, then the new driver could be implemented to present the same interface as `CHARPROC`. Other modules such as `CANONPROC` could continue to be used without modification.

Message Overview

Messages are the means of communication within a Stream. All input and output under STREAMS is based on messages. The objects passed between Streams components are pointers to messages. All messages in STREAMS use two data structures to refer to the data in the message. These data structures describe the type of the message and contain pointers to the data of the message, as well as other information. Messages are sent through a Stream by successive calls to the `put(9E)` routine of each queue in the Stream. Messages may be generated by a driver, a module, or by the Stream head.

Message Types

There are several different STREAMS messages (see Appendix B, "Message Types"). The messages differ in their intended purpose and their queueing priority. The contents of certain message types can be transferred between a process and a Stream by use of system calls.

The message types are briefly described and classified according to their queueing priority.

Ordinary Messages (also called `normal messages`):

<code>M_BREAK</code>	Request to a Stream driver to send a “break”
<code>M_CTL</code>	Control/status request used for inter-module communication
<code>M_DATA</code>	User data message for I/O system calls
<code>M_DELAY</code>	Request a real-time delay on output
<code>M_IOCTL</code>	Control/status request generated by a Stream head
<code>M_PASSFP</code>	File pointer-passing message
<code>M_PROTO</code>	Protocol control information
<code>M_SETOPTS</code>	Set options at the Stream head, sent upstream
<code>M_SIG</code>	Signal sent from a module/driver

High Priority Messages:

<code>M_COPYIN</code>	Copy in data for transparent <code>ioctl</code> s, sent downstream
<code>M_COPYOUT</code>	Copy out data for transparent <code>ioctl</code> s, sent upstream
<code>M_ERROR</code>	Report downstream error condition, sent upstream
<code>M_FLUSH</code>	Flush module queue
<code>M_HANGUP</code>	Set a Stream head hangup condition, sent upstream
<code>M_UNHANGUP</code>	Line reconnect, sent upstream when hangup reverses
<code>M_IOCACK</code>	Positive <code>ioctl(2)</code> acknowledgment
<code>M_IOCDATA</code>	Data for transparent <code>ioctl</code> s, sent downstream
<code>M_IOCNAK</code>	Negative <code>ioctl(2)</code> acknowledgment
<code>M_PCPROTO</code>	Protocol control information
<code>M_PCSIG</code>	Signal sent from a module/driver
<code>M_READ</code>	Read notification, sent downstream
<code>M_START</code>	Restart stopped device output
<code>M_STARTI</code>	Restart stopped device input
<code>M_STOP</code>	Suspend output
<code>M_STOPI</code>	Suspend input

Note – Transparent `ioctl`s, among other things, support applications developed before the introduction of STREAMS.

Expedited Data

The Open Systems Interconnection (OSI) Reference Model developed by the International Standards Organization (ISO) and International Telegraph and Telephone Consultative Committee (CCITT) provides an international standard seven-layer architecture for the development of communication protocols. SunOS 5.x adheres to this standard and also supports the Transmission Control Protocol and Internet Protocol (TCP/IP).

The OSI protocols and TCP/IP support the transport of expedited data (see note which follows) for transmission of high-priority, emergency data. This is useful for flow control, congestion control, routing, and various applications where immediate delivery of data is necessary.

Expedited data is mainly for exceptional cases and transmission of control signals. These are emergency data that are processed immediately, ahead of normal data. These messages are placed ahead of normal data in the queue, but after STREAMS high-priority messages and after any expedited data already in the queue.

Expedited data flow control is unaffected by the flow control constraints of normal data transfer. Expedited data have their own flow control because they can easily use all the system buffers if their flow is unrestricted.

Drivers and modules define separate high and low water marks for priority band data flow. (Watermarks are defined for each queue and they indicate the upper and lower limit of bytes that can be contained in the queue; see `M_SETOPTS` in “Ordinary Messages” on page 329). The default water marks for priority band data and normal data is the same. The Stream head also ensures that incoming priority band data is not blocked by normal data already in the queue. This is accomplished by associating a priority with the messages. This priority implies a certain ordering of the messages in the queue. (Message queues and priorities are discussed in the section “Message Queues and Message Priority” on page 66.)

Note – Within the STREAMS mechanism and in this guide, expedited data is also referred to as priority band data.

Message Structure

All messages are composed of one or more message blocks. A message block is a linked list of message blocks (msgb(9S)) and a data block (datab(9S)). The data buffer is a location in memory where the data of the message are stored.

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

typedef struct msgb mblk_t;

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

typedef struct datab dblk_t;
```

The field *b_band* determines where the message is placed when it is queued using the STREAMS utility routines. This field has no meaning for high priority messages and is set to zero for these messages. When a message is allocated via `allocb()`, the *b_band* field will be initially set to zero. Modules and drivers may set this field, if so desired, to a value from 0 to 255 depending on the number of priority bands needed. Lower numbers are lower priority. The kernel will incur overhead in maintaining bands if non-zero numbers are used.

Note – SunOS 5.x has `b_band` in the `msgb` struct. Some other STREAMS implementations place `b_band` in the `datab` structure. This makes the SunOS 5.x implementation more flexible since each message is independent. For shared data blocks, the `b_band` may be different in the SunOS 5.x implementation, but not in other implementations.

Message Linkage

The message block is used to link messages on a message queue, link message blocks to form a message, and manage the reading and writing of the associated data buffer. The `b_rptr` and `b_wptr` fields in the `msgb` structure are used to locate the data currently contained in the buffer. As shown in Figure 5-1, the message block (`mb1k_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. `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 overall message type, regardless of the types of the attached message blocks.

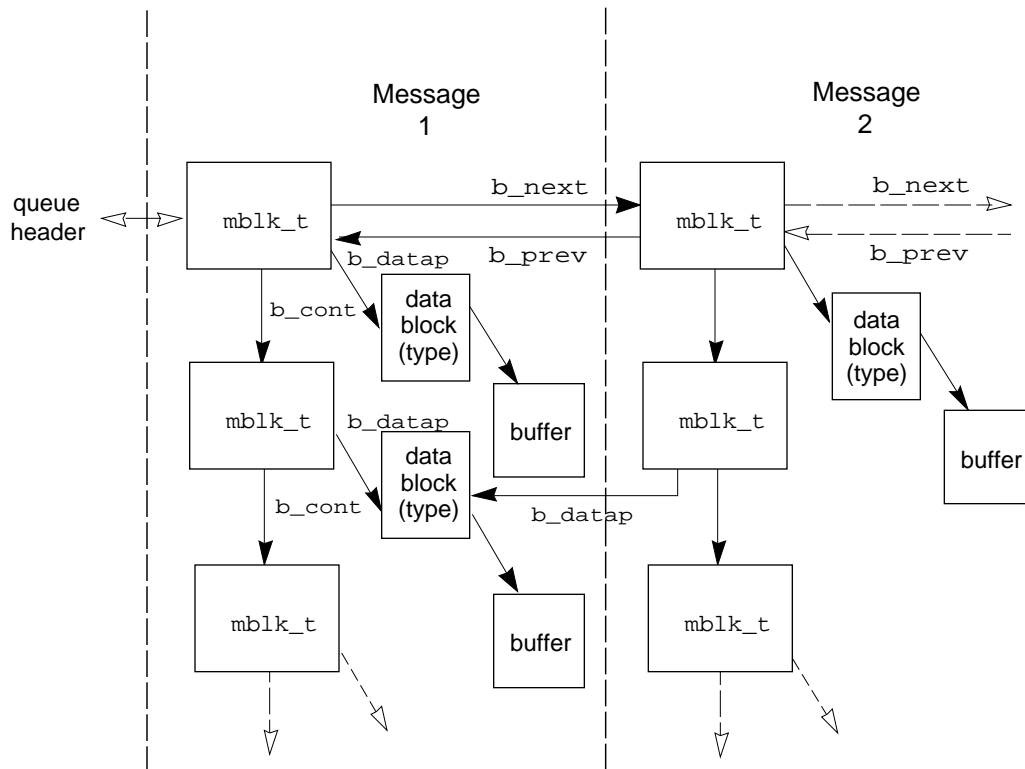


Figure 5-1 Message Form and Linkage

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 2, as shown in Figure 5-1, links to message 1.

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 5-1 illustrates the concept, but data blocks would not typically be shared by messages on the same queue). The buffer can be retransmitted, if required because of errors or timeouts, from

either protocol level without replicating the data. Data block sharing is accomplished by means of a utility routine (see `dupmsg(9F)` in “Utility Descriptions” on page 350). 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, “STREAMS Utilities”, to assist in managing messages and message queues, and to assist in other areas of module and driver development. A utility routine should always be used when operating on a message queue or accessing the message storage pool. If messages are manipulated in the queue without using the STREAMS utilities, the message ordering may become confused and lead to inconsistent results.

Note – Modules or drivers cannot modify `b_next` and `b_prev`. These fields are modified by utility routines such as `putq(9F)` and `getq(9F)`.



Caution – Not adhering to the DDI/DKI can result in panics and system crashes.

Sending/Receiving Messages

Most message types can be generated by modules and drivers. A few are reserved for the Stream head. The most commonly used messages are `M_DATA`, `M_PROTO`, and `M_PCPROTO`. These messages 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. `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.

Modules and drivers do not interact directly with any system calls except `open(2)` and `close(2)`. The Stream head handles all message translation and passing between user processes and STREAMS components. Message transfer between processes and the Stream head can occur in different forms. For example, `M_DATA` and `M_PROTO` messages can be transferred in their direct form by the `getmsg(2)` and `putmsg(2)` system calls. Alternatively, `write(2)` causes one or more `M_DATA` messages to be created from the data buffer

supplied in the call. `M_DATA` messages received at the Stream head will be consumed by `read(2)` and copied into the user buffer. As another example, `M_SIG` causes the Stream head to send a signal to a process.

Any module or driver can send any message in either direction on a Stream. However, based on their intended use in STREAMS and their treatment by the Stream head, certain messages can be categorized as upstream, downstream, or bidirectional. `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. Messages intended to be sent downstream are silently discarded if received by the Stream head.

STREAMS enables modules to create messages and pass them to neighboring modules. However, the `read(2)` and `write(2)` system calls are not sufficient to enable a user process to generate and receive all 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.

The `putmsg` system call enables a user to create messages and send them downstream. The user supplies the contents of the control and data parts of the message in two separate buffers. The `getmsg` system call retrieves `M_DATA` or `M_PROTO` messages from a Stream and places the contents into two user buffers.

The format of `putmsg` is as follows:

```
int
putmsg(
    int fd,
    const struct strbuf *ctlptr,
    const 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 high-priority message (`M_PCPROTO`) should be sent.

When a control part is present, setting *flags* to 0 generates an `M_PROTO` message. If *flags* is set to `RS_HIPRI`, an `M_PCPROTO` message is generated. Note that a *ctlptr* is translated to `M_PCPROTO` and a *dataptr* is translated to `M_DATA`.

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

The `strbuf` structure is used to describe the control and data parts of a message, and has the following interface:

```
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 `M_DATA`, `M_PROTO`, or `M_PCPROTO` messages available at the Stream head, and has the following format:

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

The arguments to `getmsg` are the same as those of `putmsg` except that the *flagsp* parameter is a pointer to an `int`.

`putpmsg()` and `getpmsg()` (see `putmsg(2)` and `getmsg(2)`) support multiple bands of data flow. They are analogous to the system calls `putmsg` and `getmsg`. The extra parameter, *band*, is the priority band of the message.

`putpmsg()` has the following interface:

```
int
putpmsg(
    int fd,
    const struct strbuf *ctlptr,
    const struct strbuf *dataptr,
    int band,
    int flags
)
```

The parameter *band* is the priority band of the message to put downstream. The valid values for *flags* are `MSG_HIPRI` and `MSG_BAND`. `MSG_BAND` and `MSG_HIPRI` are mutually exclusive. `MSG_HIPRI` generates a high-priority message (`M_PCPROTO`) and *band* is ignored. `MSG_BAND` causes an `M_PROTO` or `M_DATA` message to be generated and sent down the priority band specified by *band*. The valid range for *band* is from 0 to 255 inclusive.

The call

```
putpmsg(fd, ctlptr, dataptr, 0, MSG_BAND);
```

is equivalent to the system call

```
putmsg(fd, ctlptr, dataptr, 0);
```

and the call

```
putpmsg(fd, ctlptr, dataptr, 0, MSG_HIPRI);
```

is equivalent to the system call

```
putmsg(fd, ctlptr, dataptr, RS_HIPRI);
```

If `MSG_HIPRI` is set and *band* is nonzero, `putpmsg()` fails with `EINVAL`.

`getpmsg()` has the following format:

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

**bandp* is the priority band of the message. This system call retrieves a message from the Stream. If **flagsp* is set to `MSG_HIPRI`, `getpmsg()` attempts to retrieve a high-priority message. If `MSG_BAND` is set, `getpmsg()` tries to retrieve a message from priority band **bandp* or higher. If `MSG_ANY` is set, the first message on the Stream head read queue is retrieved. These three flags (`MSG_HIPRI`, `MSG_BAND`, and `MSG_ANY`) are mutually exclusive. On return, if a high priority message was retrieved, **flagsp* is set to `MSG_HIPRI` and **bandp* is set to 0. Otherwise, **flagsp* is set to `MSG_BAND` and **bandp* is set to the band of the message retrieved.

The call

```
int band = 0;
int flags = MSG_ANY;
getpmsg(fd, ctlptr, dataptr, &band, &flags);
```

is equivalent to

```
int flags = 0;
getmsg(fd, ctlptr, dataptr, &flags);
```

If `MSG_HIPRI` is set and **bandp* is non-zero, `getpmsg()` fails with `EINVAL`.

Control of Stream Head Processing

The `M_SETOPTS` message allows a driver or module to exercise control over certain Stream head processing. An `M_SETOPTS` message 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 A, “STREAMS Data Structures”) contained in the message. For more information on the options available in *so_flags*, see Appendix B, “Message Types”.

Six Stream head characteristics can be modified. Four characteristics correspond to fields contained in `queue` (min/max packet sizes and high/low watermarks). The other two are discussed here.

Read Options

The value for read options (*so_readopt*) corresponds to two sets of three modes a user can set via the `I_SRDOPT` ioctl (see `streamio(7)`) call. The first set of bits, `RMODEMASK`, deals with data and message boundaries:

byte-stream (RNORM)

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

message non-discard (RMSGN)

The `read(2)` call finishes when the byte count is satisfied or at a message boundary, whichever comes first. Any data remaining in the message are put back on the Stream head read queue.

message discard (RMSGD)

The `read(2)` call finishes when the byte count is satisfied or at a message boundary. Any data remaining in the message are discarded up to the message boundary.

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

The second set of bits, `RPROTMASK`, deals with the treatment of protocol messages by the `read(2)` system call:

normal protocol (`RPROTNORM`)

The `read(2)` call fails with `EBADMSG` if an `M_PROTO` or `M_PCPROTO` message is at the front of the Stream head read queue. This is the default operation protocol.

protocol discard (`RPROTDIS`)

The `read(2)` call discards any `M_PROTO` or `M_PCPROTO` blocks in a message, delivering the `M_DATA` blocks to the user.

protocol data (`RPROTDAT`)

The `read(2)` call converts the `M_PROTO` and `M_PCPROTO` message blocks to `M_DATA` blocks, treating the entire message as data.

Write Offset

If the `SO_WROFF` flag of *so_flags* is turned on, the framework uses the value for write offset (*so_wroff*) as 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 + so_wroff
```

The write offset value must be smaller than the maximum STREAMS message size, `strmsgsz` (see Appendix E, “Configuration”). In certain cases (that is, if a buffer large enough to hold the offset and the data is not currently available), the write offset might not be included in the block. To handle all possibilities, 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.

Message Queues and Message Priority

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 called by the scheduler the `service` procedure processes queued messages in a First-In-First-Out (FIFO) manner. However, expedited data support and certain conditions require that associated messages (for instance, an `M_ERROR`) reach their Stream destination as rapidly as possible. This is accomplished by associating priorities to the messages. These priorities imply a certain ordering of messages in the queue as shown in Figure 5-2. Each message has a priority band associated with it. Ordinary messages have a priority of zero. High-priority messages are high priority by nature of their message type. Their priority band is ignored. By convention, they are not affected by flow control. The `putq()` utility routine places high priority messages at the head of the message queue followed by priority band messages (expedited data) and ordinary messages.

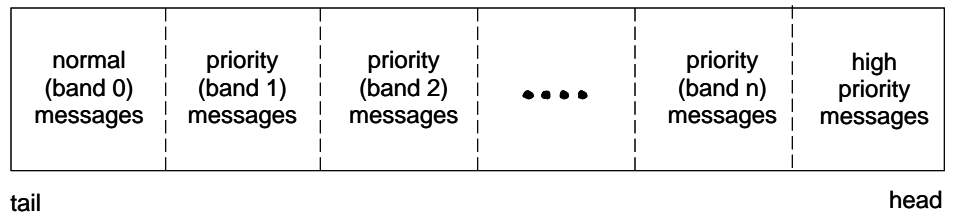


Figure 5-2 Message Ordering in a Queue

When a message is queued, it is placed after the messages of the same priority already in the queue (for instance, FIFO within their order of queueing). This affects the flow-control parameters associated with the band of the same priority. Message priorities range from 0 (normal) to 255 (highest). This provides up to 256 bands of message flow within a Stream. Expedited data can be implemented with one extra band of flow (priority band 1) of data. This is shown in Figure 5-3.

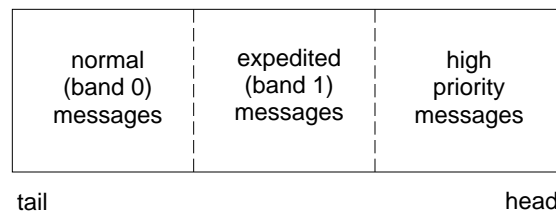


Figure 5-3 Message Ordering with One Priority Band

High-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 high priority message, if present. Service procedures must be implemented to act on high priority messages immediately. The above mechanisms—priority message queueing, absence of flow control, and immediate processing by a procedure—result in rapid transport of high priority messages between the originating and destination components in the Stream.

For example, a module may want take a message off its queue, duplicate it, and put the original message back on its queue. It may then pass the new message on to the next module. If the priority band of the new message is changed somewhere else on the Stream, the original message will be out of order in the queue. Therefore, if the reference count of the message is greater than one, it is recommended that the module copy the message via `copymsg()`, free the duplicated message, and then change the priority of the copied message. The location of `b_band` is important relating to `copymsg()`. If `b_band` is in the `msgb` structure, then copying isn't necessary. If `b_band` is in `dblkc`, then copying is necessary.

Note – A service procedure should never queue a high-priority message on its own queue, or else an infinite loop will result. The enqueueing will trigger the queue to be immediately scheduled again.

Several routines are provided to aid you in controlling each priority band of data flow. These routines are

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

The `flushband()` routine is discussed in “Flush Handling” on page 153, the `bcanputnext()` routine is discussed in “Flow Control” on page 76, and the other two routines are described below. Appendix C, “STREAMS Utilities” also has a description of these routines.

The `strqget()` routine allows modules and drivers to obtain information about a queue or particular band of the queue. This insulates the STREAMS data structures from the modules and drivers. The format of the routine is:

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

The information is returned in the `long` referenced by *valp*. The fields that can be obtained are defined by the following (defined in `<sys/stream.h>`):

<code>QLOWAT</code>	<code>/* q_lowat or qb_lowat */</code>
<code>QMAXPSZ</code>	<code>/* q_maxpsz */</code>
<code>QMINPSZ</code>	<code>/* q_minpsz */</code>
<code>QCOUNT</code>	<code>/* q_count or qb_count */</code>
<code>QFIRST</code>	<code>/* q_first or qb_first */</code>
<code>QLAST</code>	<code>/* q_last or qb_last */</code>
<code>QFLAG</code>	<code>/* q_flag or qb_flag */</code>

This routine returns 0 on success and an error number on failure.

The routine `strqset()` allows modules and drivers to change information about a queue or particular band of the queue. This also insulates the STREAMS data structures from the modules and drivers. Its format is:

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

The updated information is provided by *val*. `strqset()` returns 0 on success and an error number on failure. If the field is intended to be read-only, then the error `EPERM` is returned and the field is left unchanged. The following fields are read-only: `QCOUNT`, `QFIRST`, `QLAST`, and `QFLAG`. The use of `strqget` and `strqset` routines must be enclosed by `freezestr()` and `unfreezestr()`.

The `ioctl`s `I_FLUSHBAND`, `I_CKBAND`, `I_GETBAND`, `I_CANPUT`, and `I_ATMARK` support multiple bands of data flow. The `ioctl` `I_FLUSHBAND` allows a user to flush a particular band of messages. It is discussed in more detail in “*Flush Handling*” on page 153.

The `ioctl` `I_CKBAND` allows a user to check if a message of a given priority exists on the Stream head read queue. Its interface is:

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

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

The `ioctl` `I_GETBAND` allows a user to check the priority of the first message on the Stream head read queue. The interface is:

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

This results in the integer referenced by *prip* being set to the priority band of the message on the front of the Stream head read queue.

The `ioctl I_CANPUT` allows a user to check if a certain band is writable. Its interface is:

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

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

The field `b_flag` of the `msgb` structure can have a flag `MSGMARK` that allows a module or driver to mark a message. This is used to support TCP's (Transmission Control Protocol) ability to indicate to the user the last byte of out-of-band data. Once marked, a message sent to the Stream head causes the Stream head to remember the message. A user may check to see if the message on the front of its Stream head read queue is marked or not with the `I_ATMARK` `ioctl`. If a user is reading data from the Stream head and there are multiple messages on the read queue, and one of those messages is marked, the `read(2)` terminates when it reaches the marked message and returns the data only up to that marked message. The rest of the data may be obtained with successive reads.

The `ioctl I_ATMARK` has the following format:

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

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

The queue Structure

Service procedures, message queues, message 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` structure as described in queue (9S):

```

struct qinit      *q_qinfo; /* procs and limits for queue */
struct msgb       *q_first; /* msg que head for this queue */
struct msgb       *q_last;  /* msg queue tail for this queue */
struct queue      *q_next;   /* next queue in Stream */
struct queue      *q_link    /* to next Q for scheduling */
void              *q_ptr;    /* to module private data */
ulong             q_count;   /* number of bytes in queue */
ulong             q_flag;    /* queue state */
long              q_minpsz; /* min packet size accepted */
long              q_maxpsz; /* max packet size accepted */
ulong             q_hiwat;   /* queue high watermark */
ulong             q_lowat;   /* queue low watermark */

```

Queues are always allocated in pairs (read and write); one queue pair per a module, a driver, or a Stream head. A queue contains a linked list of messages. When a queue pair is allocated, 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 without modifying the `streamtab` and `module_info` values.

`q_count` is used in flow control calculations and is the number of bytes in messages in the queue.

Using queue Information

Modules and drives can change `q_ptr` directly. Modules and drivers can read but should not change `q_qinfo`, and `q_next`. The `strqset(9F)` utility can be used to change `q_hiwat`, `q_lowat`, `q_maxpsz`, and `q_minpsz`. Modules and drivers should use `strgget(9F)` to read `q_hiwat`, `q_lowat`, `q_maxpsz`, `q_count`, `q_first`, `q_last`, or `q_flag`.

All other accesses to fields in the `queue(9S)` structure should be made through STREAMS utility routines (see Appendix C, “STREAMS Utilities”). Modules and drivers should not change any fields not explicitly listed above. Also modules should lock their private data structures. See Chapter 13, “Multi-Threaded STREAMS” for more information on locking.

Queue Flags

Programmers using the STREAMS mechanism should be aware of the following queue flags. See `queue(9S)`.

QENAB	queue is enabled to run service procedure (on the run)
QWANTR	to read from the queue
QWANTW	to write to the queue
QFULL	queue is full
QREADR	set for all read queues
QUSE	queue has been allocated
QNOENB	do not enable the queue when data is placed on it

The qband Structure

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

```

struct qband *qb_next;           /* next band's info */
ulong        qb_count;          /* number of bytes in band */
struct msgb *qb_first;          /* beginning of band's data */
struct msgb *qb_last;           /* end of band's data */
ulong        qb_hiwat;          /* high watermark for band */
ulong        qb_lowat;          /* low watermark for band */
ulong        qb_flag;           /* flag, QB_FULL, denotes that a */
                                /* band of data flow is flow */
                                /* controlled */

```

This structure contains pointers to the linked list of messages in the queue. These pointers, `qb_first` and `qb_last`, denote the beginning and end of messages for the particular band. The `qb_count` field is analogous to the queue's `q_count`

field. However, *qb_count* only applies to the messages in the queue in the band of data flow represented by the corresponding *qband* structure. In contrast, *q_count* only contains information regarding normal and high-priority messages.

Each band has a separate high and low water mark, *qb_hiwat* and *qb_lowat*. These are initially set to the *queue*'s *q_hiwat* and *q_lowat* respectively. Modules and drivers may change these values if desired through the *strqset*(9F) function. Two flags, *QB_FULLL* and *QB_WANTW*, are defined for *qb_flag*. *QB_FULLL* denotes that the particular band is full. *QB_WANTW* indicates that someone attempted to write to the band that was flow controlled.

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

qband Flags

Programmers using the STREAMS mechanism should be aware of the following *qband* flags.

<i>QB_FULLL</i>	band is considered full
<i>QB_WANTW</i>	to write to the queue

Using qband Information

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

Drivers and modules are allowed to change the *qb_hiwat* and *qb_lowat* fields of the *qband* structure.

Drivers and modules may only read the *qb_count*, *qb_first*, *qb_last*, and *qb_flag* fields of the *qband* structure.

Only the fields listed previously may be referenced at all. There are fields in the structure that are reserved and are thus not documented.

Figure 5-4 shows a queue with two extra bands of flow.

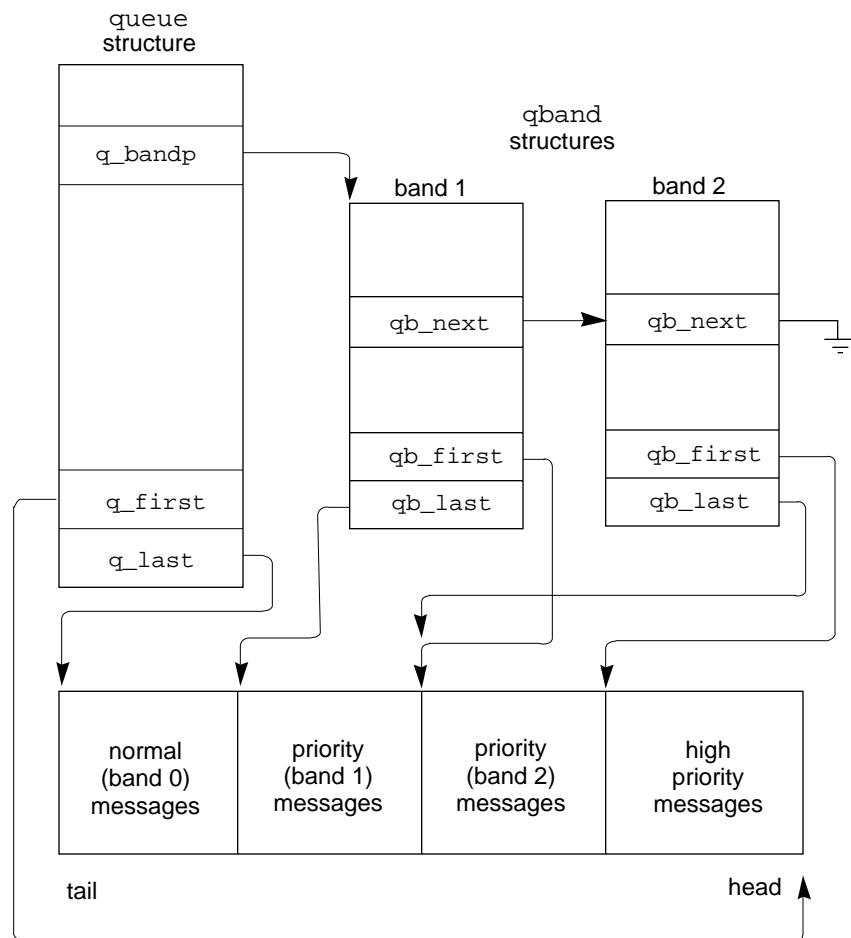


Figure 5-4 Data Structure Linkage

Message Processing

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

The general processing flow when both procedures are present is as follows:

1. A message is received by the `put` procedure associated with queue, where some processing may be performed on the message.
2. The `put` procedure places the message in the queue by use of the `putq()` utility routine for the `service` procedure to perform further processing later.
3. `putq()` places the message in the queue based on its priority.
4. Then, `putq()` makes the queue ready for execution by the STREAMS scheduler following all other queues currently scheduled.
5. When the system goes from kernel mode to user mode, the STREAMS scheduler calls the `service` procedure.
6. The `service` procedure gets the first message (*q_first*) from the message queue by using the `getq()` utility.
7. The `service` procedure processes the message and passes it to the `put` procedure of the next queue with `putnext()`.
8. The `service` procedure gets the next message and processes it. This processing continues until the queue is empty or flow control blocks further processing. The `service` procedure returns to the caller.



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

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

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

More typically, `put` procedures will, at a minimum, process high priority messages 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.

Note – The presence of a `service` procedure is mandatory if the flow control mechanism is to be utilized by the queue. If you don't implement flow control, it is possible to overflow queues and hang the system.

Flow Control

The STREAMS flow control mechanism is voluntary and operates between the two nearest queues in a Stream containing `service` procedures (see Figure 5-5). Messages are generally held on a queue only if a `service` procedure is present in the associated queue.

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

Flow control operates as follows:

1. Each time a STREAMS message handling routine (for example, `putq`) adds or removes a message from a message queue, the limits are checked. STREAMS calculates the total size of all message blocks (`bp->b_wptr - bp->b_rptr`) on the message queue.
2. The total is compared to the queue high water and low water mark values. If the total exceeds the high watermark value, an internal full indicator is set for the queue. The operation of the `service` procedure in this queue is not affected if the indicator is set, and the `service` procedure continues to be scheduled.
3. The next part of flow control processing occurs in the nearest preceding queue that contains a `service` procedure. In Figure 5-5, if D is full and C has no `service` procedure, then B is the nearest preceding queue.

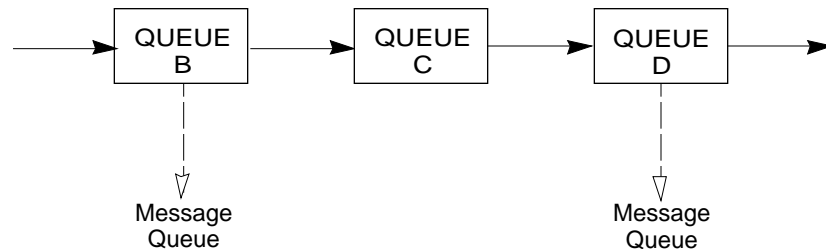


Figure 5-5 Flow Control

4. The `service` procedure in B uses a STREAMS utility routine, `canputnext()`, to see if a queue ahead is marked full. If messages cannot be sent, the scheduler blocks the `service` procedure in B from further execution. B remains blocked until the low watermark of the full queue, D, is reached.
5. While B is blocked, any messages except high priority messages arriving at B will accumulate on its message queue (recall that high priority messages are not subject to flow control). Eventually, B may reach a full state and the full condition will propagate back to the preceding module in the Stream.

6. When the `service` procedure processing on D causes the message block total to fall below the low watermark, the full indicator is turned off. Then, STREAMS automatically schedules the nearest preceding blocked queue (B in this case), getting things moving again. This automatic scheduling is known as back-enabling a queue.

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

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

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

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

This is the general flow control processing of ordinary messages:

1. Retrieve the message at the head of the queue with `getq()`.
2. Determine if the message type is high priority and not to be processed here.
3. If so, pass the message to the `put` procedure of the following queue with `putnext()`.
4. Use `canputnext()` to determine if messages can be sent onward.
5. If messages should not be forwarded, put the message back in the queue with `putbq()` and return from the procedure.
6. Otherwise, process the message.

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

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

```
        putbq()  
        return  
    }
```

Expedited data have their own flow control with the same general processing as that of ordinary messages. `bcanputnext(9F)` is used to provide modules and drivers with a way to test flow control in the given priority band. It returns 1 if a message of the given priority can be placed in the queue. It returns 0 if the priority band is flow controlled. If the band does not yet exist in the queue in question, the routine returns 1.

If the band is flow controlled, the higher bands are not affected. However, the same is not true for lower bands. The lower bands are also stopped from sending messages. If this didn't take place, the possibility would exist where lower priority messages would be passed along ahead of the flow controlled higher priority ones.

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

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

A service procedure continues processing messages from its queue until `getq()` returns NULL. When an ordinary message is queued by `putq()`, `putq()` will cause the service procedure to be scheduled only if the queue was previously empty, and a previous `getq()` call returns NULL (that is, the `QWANTR` flag is set). If there are messages in 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 (for example, no buffers), it must ensure it will return later (for example, by use of `bufcall()` utility routine) or it must discard all messages in the 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()`.

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

`putbq()` replaces messages at the beginning of the appropriate section of the message queue in accordance with their priority. 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.

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

This also applies to `putbq()` and `insq()`. If an attempt is made to insert a message out of order in a queue via `insq()`, the message is not inserted and the routine fails.

`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)`.

The service procedure is written using the following algorithm:

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

If the module or driver is unconcerned with priority bands, the algorithm is the same as described in the previous paragraphs, except that `canputnext(q)` is substituted for the `bcanputnex()` call.

Driver upstream flow control is explained next as an example. 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 blockages. 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 `canputnext`), it sends the message with `putnext()`. Otherwise, it calls `putq()` to queue the message. The message waits in the message queue (possibly with queue length checked when new messages are queued by the interrupt routine) until the upstream queue becomes clear. When the blockage abates, STREAMS back-enables the driver read `service` procedure. The `service` procedure sends the messages upstream using `getq()` and `canputnext()`, as described previously. This is similar to `looprsrv()` (See “Loop-Around Driver” section of Chapter 9, “Drivers” 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. `qenable()` might also 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 module calls `qenable()` to cause its `service` procedure to run.

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

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

Service Interfaces

STREAMS provides the means to implement a service interface between any two components in a Stream, and between a user process and the topmost module in the Stream. A service interface is defined at the boundary between a service user and a service provider (see Figure 5-7). A *service interface* is a set of primitives and the rules that define a service and the allowable state transitions that result as these primitives are passed between the user and the provider. These rules are typically represented by a state machine. In STREAMS, the service user and provider are implemented in a module, driver, or user process. The primitives are carried bidirectionally between a service user and provider in `M_PROTO` and `M_PCPROTO` messages.

`PROTO` messages (`M_PROTO` and `M_PCPROTO`) can be multi-block, with the second through last blocks of type `M_DATA`. The first block in a `PROTO` message contains the control part of the primitive in a form agreed upon by the user and provider. The block is not intended to carry protocol headers. (Although its use is not recommended, upstream `PROTO` messages can have multiple `PROTO` blocks at the start of the message. `getmsg(2)` 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 (upstream and downstream) on a Stream and between a Stream and a user process. `putmsg(2)` and `getmsg(2)` system calls are analogous, respectively, to `write(2)` and `read(2)` except that the former allow both data and control parts to be (separately) passed, and they retain the message boundaries across the user-Stream interface. `putmsg(2)` and `getmsg(2)` separately copy the control part (`M_PROTO` or `M_PCPROTO` block) and data part (`M_DATA` blocks) between the Stream and user process.

An `M_PCPROTO` message is normally used to acknowledge primitives composed of other messages. `M_PCPROTO` 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(2)`.

A STREAMS message format has been defined to simplify the design of service interfaces. The `getmsg(2)` and `putmsg(2)` system calls are available for sending messages downstream and receiving messages that are available at the Stream head.

This section describes these system calls in the context of a service interface example. First, a brief overview of STREAMS service interfaces is presented.

Service Interface Benefits

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 functions. 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 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 5-6 demonstrates this.

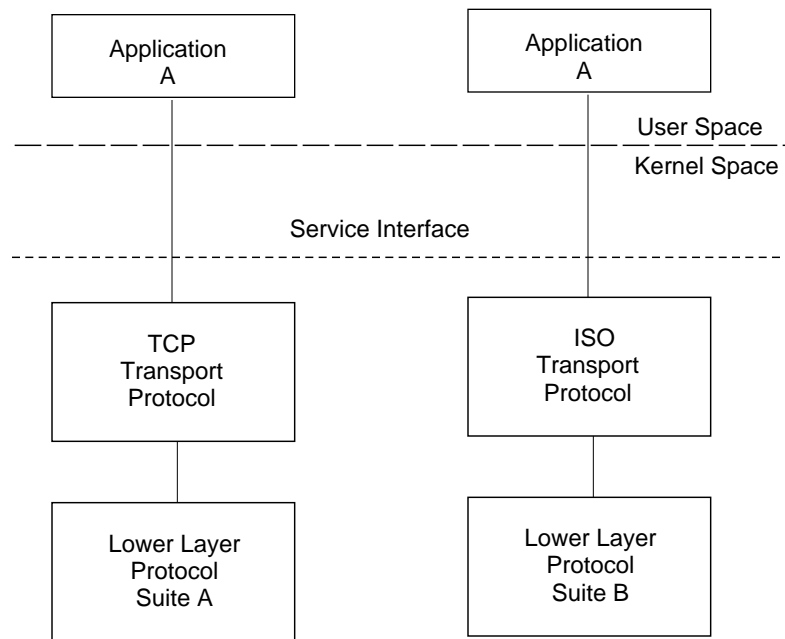


Figure 5-6 Protocol Substitution

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 the following figure.

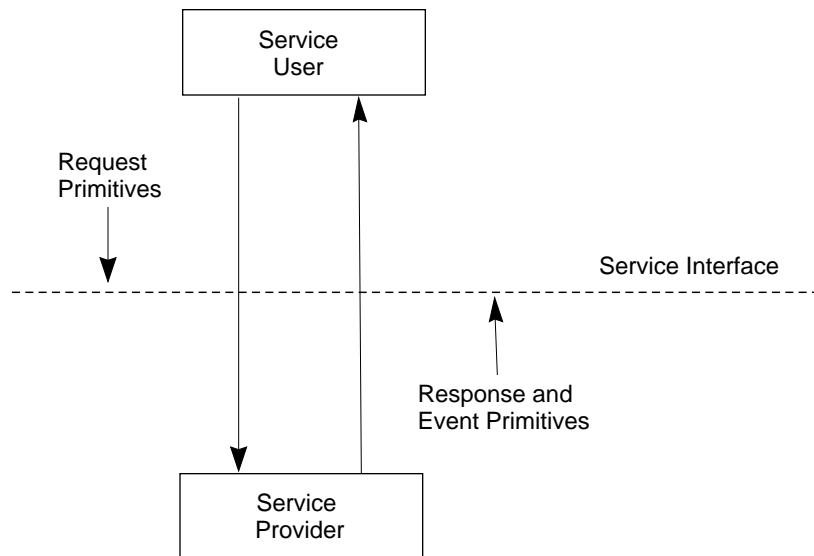


Figure 5-7 Service Interface

Typically, a user requests of a service provider using some well-defined service primitive. Responses and event indications are also passed from the provider to the user using service primitives.

Each service interface primitive is a distinct STREAMS message that has two parts; a control part and a data part. The control part contains information that identifies the primitive and includes all necessary parameters. The data part contains user data associated with that primitive.

An example of a service interface primitive is a transport protocol connect request. This primitive requests the transport protocol service provider to establish a connection with another transport user. The parameters associated with this primitive 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.

Service Interface Library Example

The service interface library example presented here 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 data to a remote user,
- receive data from a remote user, and
- close the Stream connected to the provider

First, the structure and constant definitions required by the library are shown in the following example. 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 {
    long    PRIM_type;          /* bind request */
    long    BIND_addr;          /* always BIND_REQ */
    /* addr to bind */
};
struct unitdata_req {
    long    PRIM_type;          /* unitdata request */
    long    DEST_addr;          /* always UNITDATA_REQ */
    /* destination addr */
};

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

```
};

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

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

/* union of all primitives */
union primitives {
    long        type;
    struct bind_req    bind_req;
    struct unitdata_req    unitdata_req;
    struct ok_ack    ok_ack;
    struct error_ack    error_ack;
    struct unitdata_ind    unitdata_ind;
};

/* header files needed by library */
#include <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 data to the specified destination address. It does not require an acknowledgment from the provider. |

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

The defined structures describe the contents of the control part of each service interface message passed between the service user and service provider. The first field of each control part defines the type of primitive being passed.

Accessing the Service Provider

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

Code Example 5-1 *inter_open*

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

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

    /* send bind request msg down stream */

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

    if (putmsg(fd, &ctlbuf, NULL, 0) < 0) {
```

Code Example 5-1 `inter_open`

```
        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 high priority message.

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

Code Example 5-2 Service Provider Example

```
/* 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) {
```

Code Example 5-2 Service Provider Example

```

default:
    close(fd);
    errno = EPROTO;
    return(-1);

case OK_ACK:
    return(fd);

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

```

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

The acknowledgment primitives are defined as high priority messages. Messages are queued in a first-in-first-out manner within their priority at the Stream head; high priority messages are placed at the front of the Stream head queue followed by priority band messages and ordinary messages. The STREAMS mechanism allows only one high priority message per Stream at the Stream head at one time. Any additional high priority messages will be discarded upon reaching the Stream head. (There can be only one high priority message present on the Stream head read queue at any time.) High priority messages are particularly suitable for acknowledging service requests when the acknowledgment should be placed ahead of any other messages at the Stream head.

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

Because neither acknowledgment primitive contains a data part, the *dataptr* argument to `getmsg` is set to `NULL`. The *flagsp* argument points to an integer containing the value `RS_HIPRI`. This flag indicates that `getmsg` should wait for a STREAMS high priority message before returning. It is set because you want to catch the acknowledgment primitives that are priority messages. Otherwise, if the flag is zero, the first message is taken. With `RS_HIPRI` set, even if a normal message is available, `getmsg` will block until a high 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 Provider

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

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

The routine closes the given file descriptor. This causes the protocol driver to free any resources associated with that Stream. For example, the driver 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 Data to Service Provider

The third routine, *inter_snd*, passes data to the service provider for transmission to the user at the address specified in *addr*. The data to be transmitted are 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(int fd, char *buf, int len, long *addr)
{
    struct strbuf ctlbuf;
    struct strbuf databuf;
    struct unitdata_req unitdata_req;

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

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

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

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

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

Receiving Data

The final routine in this example, *inter_rcv*, retrieves the next available data. *buf* points to a buffer where the data should be stored, *len* indicates the size of that buffer, and *addr* points to a long integer where the source address of the data

will be placed. On successful completion, *inter_rcv* returns the number of bytes of retrieved data; on failure, it returns -1 and an appropriate UNIX system error value.

Figure 5-8 Receiving Data

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

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

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

getmsg is called to retrieve the data indication primitive, where that primitive contains both a control and data part. The control part consists of a *unitdata_ind* structure that identifies the primitive type and the source address of the data sender. The data part contains the data itself.

In *ctlbuf*, *buf* 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 integer pointed at by *flagsp* in the `getmsg` call is set to zero, indicating that the next message should be retrieved from the Stream head, regardless of its priority. Data 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 data. If both buffers are large enough, `getmsg` will process the data indication and return 0, indicating that a full message was retrieved successfully. However, if neither buffer is 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 (if in message non-discard mode), 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 data indication (`UNITDATA_IND` in the example). The source address is then set and the number of bytes of data is returned.

The example presented is a simplified service interface. 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. For simplicity, this example did not also consider expedited data.

Module Service Interface Example

The following example is part of a module that illustrates the concept of a service interface. The module implements a simple service interface and mirrors the service interface library example. The following rules pertain to service interfaces:

- Modules and drivers that support a service interface must act upon all `PROTO` messages and not pass them through.

- Modules 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 high priority messages are not subject to flow control and forward them accordingly.

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 {
    long    PRIM_type;          /* bind request */
    long    BIND_addr;          /* always BIND_REQ */
    /* addr to bind*/
};
struct unitdata_req {
    long    PRIM_type;          /* unitdata request */
    long    DEST_addr;          /* always UNITDATA_REQ */
    /* dest addr */
};
struct ok_ack {
    /* ok acknowledgment */
```

```

    long        PRIM_type;           /* always OK_ACK */
};
struct error_ack {                   /* error acknowledgment */
    long        PRIM_type;           /* always ERROR_ACK */
    long        UNIX_error;          /* UNIX system error code*/
};
struct unitdata_ind {                /* unitdata indication */
    long        PRIM_type;           /* always UNITDATA_IND */
    long        SRC_addr;            /* source addr */
};

union primitives {                   /* union of all primitives */
    long        type;
    struct bind_req bind_req;
    struct unitdata_req unitdata_req;
    struct ok_ack ok_ack;
    struct error_ack error_ack;
    struct unitdata_ind unitdata_ind;
};
struct dgproto {                     /* structure minor device */
    short state;                     /* current provider state */
    long addr;                        /* net address */
};

/* Provider states */
#define IDLE 0
#define BOUND 1

```

In general, the M_PROTO or M_PCPROTO block is described by a data structure containing the service interface information. In this example, union primitives is that structure.

The module recognizes two commands:

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

UNITDATA_REQ Send data to the specified address.

The module generates three messages:

OK_ACK A positive acknowledgment (ack) of BIND_REQ.

ERROR_ACK A negative acknowledgment (nak) of BIND_REQ.

UNITDATA_IND Data from the network have been received.

The acknowledgment of a `BIND_REQ` informs the user that the request was syntactically correct (or incorrect if `ERROR_ACK`). The receipt of a `BIND_REQ` is acknowledged with an `M_PCPROTO` to 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:

<i>state</i>	current state of the service provider <code>IDLE</code> or <code>BOUND</code>
<i>addr</i>	network address that has been bound to this Stream

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

Service Interface Procedure

The write put procedure is:

```
static int protowput(queue_t *q, mblk_t *mp)
{
    union primitives *proto;
    struct dgproto *dgproto;
    int err;
    dgproto = (struct dgproto *) q->q_ptr; /* priv data struct */
    switch (mp->b_datap->db_type) {
    default:
        /* don't understand it */
        mp->b_datap->db_type = M_ERROR;
        mp->b_rptr=mp->b_wptr=mp->b_datap->db_base;
        *mp->b_wptr++ = EPROTO;
        qreply(q, mp);
        break;
    case M_FLUSH: /* standard flush handling goes here ... */
        break;
    case M_PROTO:
        /* Protocol message -> user request */
        proto = (union primitives *) mp->b_rptr;
        switch (proto->type) {
        default:
            mp->b_datap->db_type = M_ERROR;
            mp->b_rptr=mp->b_wptr=mp->b_datap->db_base;
            *mp->b_wptr++ = EPROTO;
            qreply(q, mp);
            return;
        }
    }
}
```

```

case BIND_REQ:
    if (dgproto->state != IDLE) {
        err = EINVAL;
        goto error_ack;
    }
    if (mp->b_wptr - mp->b_rptr !=
        sizeof(struct bind_req)) {
        err = EINVAL;
        goto error_ack;
    }
    if (err = chkaddr(proto->bind_req.BIND_addr))
        goto error_ack;
    dgproto->state = BOUND;
    dgproto->addr = proto->bind_req.BIND_addr;
    mp->b_datap->db_type = M_PCPROTO;
    proto->type = OK_ACK;
    mp->b_wptr=mp->b_rptr+sizeof(structok_ack);
    qreply(q, mp);
    break;
error_ack:
    mp->b_datap->db_type = M_PCPROTO;
    proto->type = ERROR_ACK;
    proto->error_ack.UNIX_error = err;
    mp->b_wptr=mp->b_rptr+sizeof(structerror_ack);
    qreply(q, mp);
    break;
case UNITDATA_REQ:
    if (dgproto->state != BOUND)
        goto bad;
    if (mp->b_wptr - mp->b_rptr !=
        sizeof(struct unitdata_req))
        goto bad;
    if(err=chkaddr(proto->unitdata_req.DEST_addr))
        goto bad;
    putq(q, mp);
    /* start device or mux output ... */
    break;
bad:
    freemsg(mp);
    break;
    }
}
return(0);
}

```


The write `put` procedure 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. If all is well, the message is put in the queue, and the lower half of the driver is started.

If the write `put` procedure receives a message type that it does not understand, either a bad `b_datap->db_type` or bad `proto->type`, the message is converted into an `M_ERROR` message and sent upstream.

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

Message Allocation and Freeing

The `allocb(9F)` utility routine is used to allocate a message and the space to hold the data for the message. `allocb()` returns a pointer to a message block containing a data buffer of at least the size requested, providing there is enough memory available. It returns null on failure. Note that `allocb()` always returns 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 *msgb* and *datab*), which is the start of the memory location for the data.

`allocb()` may return a buffer larger than the size requested. If `allocb()` indicates buffers are not available (`allocb()` fails), the `put/service` procedure may not block to wait for a buffer to become available. Instead, the `bufcall()` utility can be used to defer processing in the module or the driver until a buffer becomes available.

If message space allocation is done by the `put` procedure and `allocb()` fails, the message is usually discarded. If the allocation fails in the `service` routine, the message is returned to the queue. `bufcall()` is called to enable to the service routine when a message buffer becomes available, and the `service` routine returns.

The `freeb()` utility routine releases the message block descriptor and the corresponding data block, if the reference count (see `datab` structure) is equal to 1. If the reference count exceeds 1, the data block is not released.

The `freemsg()` utility routine releases all message blocks in a message. It uses `freeb()` to free all message blocks and corresponding data blocks.

In the following example, `allocb()` is used by the `bappend` subroutine that 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 int bappend(mblk_t **bpp, int ch)
{
    mblk_t *bp;

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

bappend receives a pointer to a message block 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()`.

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.

The next example subroutine `modwput` processes all the message blocks in any downstream data (type `M_DATA`) messages. `freemsg()` frees messages.

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

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

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

                bp->b_rptr++;
                continue;

            newblk:
                if (nmp == NULL)
                    nmp = nbp;
                else { /* link msg blk to tail of nmp */
                    linkb(nmp, nbp);
                    nbp = NULL;
                }
            }
        }
    }
```

```
    }
    if (nmp == NULL)
        nmp = nbp;
    else
        linkb(nmp, nbp);
    freemsg(mp); /* de-allocate message */
    if (nmp)
        putnext(q, nmp);
    break;
}
}
```

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

Recovering From No Buffers

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

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

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

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



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

Two examples are provided. The first is a device-receive-interrupt handler:
Code Example 5-3 Device Interrupt handler

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

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

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

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

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()*. If *allocb()* fails, *dev_re_load* uses *bufcall()* to call itself when STREAMS determines a buffer is available. *id* is saved as the return value from *bufcall()* to be used later by *unbufcall()* prior to closing the driver. This is important to be aware of as a system crash due to a callback that still has a *bufcall()* request pending is very difficult to track down. See Chapter 13, "Multi-Threaded STREAMS" for more information on the uses of *unbufcall()*. These references are protected by MT locks.

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 getting hung.

The second example is a write service procedure, *mod_wsrv*, which needs to prefix each output message with a header.

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

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

In this previous example, *mod_wsrv* illustrates a case for potential deadlock. If `allocb()` fails, *mod_wsrv* tends to recover without loss of data and calls `bufcall()`. In this case, the routine passed to `bufcall()` is `qenable()`. When a buffer is available, the service procedure will be automatically re-enabled. Before exiting, the current message is put back in the queue. This example deals with `bufcall()` failure by resorting to the `timeout()` operating system utility routine. `timeout()` will schedule the given function to be run with the given argument in the given number of clock cycles (there are 1,000,000 microseconds per second). In this example, if `bufcall()` fails, the system will run `qenable()` after two seconds have passed.

Releasing Callback Requests

When `allocb()` fails and a call is made to `bufcall()`, a callback is pending until a buffer is actually returned. Since this callback is an asynchronous process, it must be released before all processing is complete. To release this queued event, use `unbufcall()`.

Pass the *id* returned from `bufcall()` to the `unbufcall()` routine. Then you can close the driver in the normal way. If this sequence of `unbufcall()` and `xxclose()` is not followed, then a situation exists where the callback can occur and the driver will be closed. This is one of the most difficult types of bugs to track down during the debugging stage.

Extended STREAMS Buffers

Some hardware using the STREAMS mechanism supports memory-mapped I/O (see `mmap()`) that allows the sharing of buffers between users, kernel, and the I/O card.

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

In a typical network arrangement, data is received from the network by the I/O card. The controller reads the block of data into the card's internal buffer. It interrupts the host computer to denote that data have arrived. The

STREAMS driver gives the controller the kernel address where the data block is to go and the number of bytes to transfer. After the controller has read the data into its buffer and verified the checksum, it copies the data into main memory to the address specified by the DMA (direct memory access) memory address. Once in the kernel space, the data is packaged into message blocks and processed in the usual manner.

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

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

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

- **Allocation** - If the drivers are to use DARAM without wasting STREAMS resources and without being dependent on upstream modules, a data and message block can be allocated without an allocated data buffer. The routine to use is called `esballoc(9F)`. This returns a message block and data block without an associated STREAMS buffer. Rather, the buffer used is the one supplied by the caller in the buffer passed in.
- **Freeing** - Each driver using non-STREAMS resources in a STREAMS environment must fully manage those resources, including freeing them. However, to make this as transparent as possible, a driver-dependent routine is executed in the event `freeb()` is called to free a message and data block with an attached non-STREAMS buffer.

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

The free routine should not reference any dynamically allocated data structures that become freed when the driver is closed, as messages can exist in a Stream after the driver is closed. For example, when a Stream is closed, the driver close routine is called and its private data structure may be deallocated. If the driver sends a message created by `esballoc` upstream, that message may still be on the Stream head read queue. When the Stream head read queue is flushed, the message is freed and a call is made to the driver's free routine after the driver has been closed.

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

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

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

The method by which `free_func` is called is implementation-specific. Do not assume that `free_func` will or will not be called directly from STREAMS utility routines like `freeb()`. The `free_func` function must not call another module's put procedure nor attempt to acquire a private module lock that may be held by another thread across a call to a STREAMS utility routine which could free a message block. Otherwise, the possibility for lock recursion and/or deadlock exists.

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

esballoc Example

This skeletal example (which will not compile) shows how extended buffers are managed in the multithreaded environment. The driver maintains a pool of special memory which is *esballoc*'ed. The allocator free routine uses the queue struct assigned to the driver, or some other queue private data, so the allocator and the close routine need to coordinate to ensure that no outstanding *esballoc*'ed memory blocks remain after the close. The special memory blocks are of type *ebm_t*, the counter is *ebm*, the mutex *mp* and the condition variable *cvp* are used to implement the coordination:

Code Example 5-4 *esballoc Example*

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

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

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

Code Example 5-4 esballoc Example

```
    mutex_exit(&mp);
}

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

    mutex_exit(&mp);}
```


Input/Output Polling

This chapter describes the synchronous polling mechanism and asynchronous event notification within STREAMS. Also discussed is how a Stream can be a controlling terminal.

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

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

Instead of waiting for an event to occur, a user process may want to monitor one or more Streams while processing other data. It can do so by issuing the `I_SETSIG ioctl(2)` command, specifying one or more Streams and events (as with `poll(2)`). This `ioctl` does not block the process and force the user process to wait for the event but returns immediately and issues a signal when an event occurs. The process must request `signal(2)` to catch the resultant `SIGPOLL` signal.

If any selected event occurs on any of the selected Streams, STREAMS will send `SIGPOLL` to all associated requesting processes. However, the process(es) will not know which event occurred, nor on what Stream the event occurred. A process that issues the `I_SETSIG` can get more detailed information by issuing a `poll` after it detects the event.

Synchronous Input/Output

The `poll(2)` system call provides a mechanism to identify 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. The types of events that can be polled are `POLLIN`, `POLLRDNORM`, `POLLRDBAND`, `POLLPRI`, `POLLOUT`, `POLLWRNORM`, `POLLWRBAND`:

<code>POLLIN</code>	A message other than an <code>M_PCPROTO</code> is at the front of the Stream head read queue. This event is maintained for compatibility with the previous releases of Solaris.
<code>POLLRDNORM</code>	A normal (non-priority) message is at the front of the Stream head read queue.
<code>POLLRDBAND</code>	A priority message (<code>band > 0</code>) is at the front of the Stream head queue.
<code>POLLPRI</code>	A high priority message (<code>M_PCPROTO</code>) is at the front of the Stream head read queue.
<code>POLLOUT</code>	The normal priority band of the queue is writable (not flow controlled).
<code>POLLWRNORM</code>	The same as <code>POLLOUT</code> .
<code>POLLWRBAND</code>	A priority band greater than 0 of a queue downstream.

Some of the events may not be applicable to all file types. For example, it is not expected that the `POLLPRI` event will be generated when polling a regular file. `POLLIN`, `POLLRDNORM`, `POLLRDBAND`, and `POLLPRI` are set even if the message is of zero length.

The `poll` system call 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. `poll(2)` takes the following arguments:

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

The following example shows the use of `poll`. Two separate minor devices of the communications driver are opened, thereby establishing two separate Streams to the driver. The *pollfd* entry is initialized for each device. Each Stream is polled for incoming data. If data arrive on either Stream, data is read and then written back to the other Stream.

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

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

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

The variable *pollfds* is declared as an array of the *pollfd* structure that 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 continues to process incoming data as follows:

Code Example 6-1 Polling

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

while (1) {
    /* poll and use -1 timeout (infinite) */
    if (poll(pollfds, NPOLL, -1) < 0) {
        perror("poll failed");
        exit(3);
    }
    for (i = 0; i < NPOLL; i++) {
        switch (pollfds[i].revents) {

            default:                /* default error case */
                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)
                /*
                 * write loses data if flow control
```


Code Example 6-1 Polling

```

                                * prevents the transmit at this time
                                */
                                if (write(pollfds[(i+1) % NPOLL].fd buf,
                                    count) != count)
                                    fprintf(stderr, "writer lost
data");
                                break;
                            }
                        }
                    }
}

```

The user specifies the polled events by setting the *events* field of the *pollfd* structure to *POLLIN*. This 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 the *poll* system call specifies the number of entries in the *pollfds* array (2 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 value available on that system. If the value of *timeout* is 0, *poll* returns immediately. Here, the value of *timeout* is -1, specifying that *poll* should block until a requested event occurs or until the call is interrupted.

If the *poll* call succeeds, the program looks at each entry in the *pollfds* array. If *revents* is set to 0, no event has occurred on that file descriptor. If *revents* is set to *POLLIN*, incoming data is available. 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 *POLLIN* was the only requested event. The following are *poll* error events:

<i>POLLERR</i>	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. This event and <code>POLLOUT</code> are mutually exclusive; a Stream can't be writable if a hangup has occurred.
POLLNVAL	The specified file descriptor is not associated with an open Stream.

These events 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.

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` (or `O_NONBLOCK`, see note) flag set. The `write` will not be able to send all the data if flow control is exerted and `O_NDELAY` (`O_NONBLOCK`) is set. 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 the `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`.

Note – For conformance with the IEEE operating system interface standard, POSIX, it is recommended that new applications use the `O_NONBLOCK` flag, whose behavior is the same as that of `O_NDELAY` unless otherwise noted.

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

Asynchronous Input/Output

The `poll` system call enables a user to monitor multiple Streams synchronously. The `poll(2)` 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.

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 events for the `ioctl I_SETSIG`. These are similar to those described for `poll(2)`.

<code>S_INPUT</code>	A message other than an <code>M_PCPROTO</code> is at the front of the Stream head read queue. This event is maintained for compatibility with the previous releases of Solaris.
<code>S_RDNORM</code>	A normal (non-priority) message is at the front of the Stream head read queue.
<code>S_RDBAND</code>	A priority message (<code>band > 0</code>) is at the front of the Stream head read queue.
<code>S_HIPRI</code>	A high priority message (<code>M_PCPROTO</code>) is present at the front of the Stream head read queue.
<code>S_OUTPUT</code>	A write queue for normal data (priority band = 0) is no longer full (not flow controlled). This notifies a user that there is room on the queue for sending or writing normal data downstream.
<code>S_WRNORM</code>	The same as <code>S_OUTPUT</code> .
<code>S_WRBAND</code>	A priority band greater than 0 of a queue downstream exists and is writable. This notifies a user that there is room on the queue for sending or writing priority data downstream.
<code>S_MSG</code>	An <code>M_SIG</code> or <code>M_PCSIG</code> message containing the <code>SIGPOLL</code> flag has reached the front of Stream head read queue.
<code>S_ERROR</code>	An <code>M_ERROR</code> message reaches the Stream head.
<code>S_HANGUP</code>	An <code>M_HANGUP</code> message reaches the Stream head.
<code>S_BANDURG</code>	When used with <code>S_RDBAND</code> , <code>SIGURG</code> is generated instead of <code>SIGPOLL</code> when a priority message reaches the front of the Stream head read queue.

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

Signals

STREAMS allows modules and drivers to cause a signal to be sent to user process(es) through an `M_SIG` or `M_PCSIG` message. The first byte of the message specifies the signal for the Stream head to generate. If the signal is not `SIGPOLL` (see `signal(2)`), the signal is sent to the process group associated with the Stream. If the signal is `SIGPOLL`, the signal is only sent to processes that have registered for the signal by using the `I_SETSIG` `ioctl(2)`.

An `M_SIG` message can be used by modules or drivers that wish to insert an explicit in-band signal into a message Stream. For example, this 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` message reaches the head of the Stream head read queue, a signal is generated and the `M_SIG` message is removed. This leaves the service interface message as the next message to be processed by the user. Use of the `M_SIG` message is typically defined as part of the service interface of the driver or module.

Extended Signals

To enable a process to obtain the band and event associated with `SIGPOLL` more readily, STREAMS supports extended signals. For the given events, a special code is defined in `<sys/signinfo.h>` that describes the reason `SIGPOLL` was generated. The following table describes the data available in the `signinfo_t` structure passed to the signal handler.

Table 6-1 Data in `signinfo_t` structure

Event	si_signo	si_code	si_band	si_errno
S_INPUT	SIGPOLL	POLL_IN	band readable	unused
S_OUTPUT	SIGPOLL	POLL_OUT	band writable	unused
S_MSG	SIGPOLL	POLL_MSG	band signaled	unused
S_ERROR	SIGPOLL	POLL_ERR	unused	Stream error
S_HANGUP	SIGPOLL	POLL_HUP	unused	unused
S_HIPRI	SIGPOLL	POLL_PRI	unused	unused

Stream as a Controlling Terminal

Job Control

An overview of Job Control is provided here because it interacts with the STREAMS-based terminal subsystem. More information on Job Control may be obtained from the following manual pages: `exit(2)`, `getpgid(2)`, `getpgrp(2)`, `getsid(2)`, `kill(2)`, `setpgid(2)`, `setpgrp(2)`, `setsid(2)`, `sigaction(2)`, `signal(2)`, `sigsend(2)`, `termios(2)`, `waitid(2)`, `waitpid(3C)`, `signal(5)`, and `termio(7)`.

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

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

The following list defines terms associated with Job Control:

- **Background Process group** - A process group that is a member of a session that established a connection with a controlling terminal and is not the foreground process group.
- **Controlling Process** - A session leader that established a connection to a controlling terminal.
- **Controlling Terminal** - A terminal that is associated with a session. Each session may have at most one controlling terminal associated with it and a controlling terminal may be associated with at most one session. Certain input sequences from the controlling terminal cause signals to be sent to the process groups in the session associated with the controlling terminal.

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

The following signals manage Job Control: (see also `signal(5)`)

<code>SIGCONT</code>	Sent to a stopped process to continue it.
<code>SIGSTOP</code>	Sent to a process to stop it. This signal cannot be caught or ignored.
<code>SIGTSTP</code>	Sent to a process to stop it. It is typically used when a user requests to stop the foreground process.
<code>SIGTTIN</code>	Sent to a background process to stop it when it attempts to read from the controlling terminal.
<code>SIGTTOU</code>	Sent to a background process to stop it when one attempts to write to or modify the controlling terminal.

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

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

<code>SUSP</code> character	A user defined character that, when typed, causes the line discipline module to request that the Stream head send a <code>SIGTSTP</code> signal to the foreground process with an <code>M_PCSIG</code> message, which by default stops the members of that group. If the value of <code>SUSP</code> is zero, the <code>SIGTSTP</code> signal is not sent, and the <code>SUSP</code> character is disabled.
<code>TOSTOP</code> flag	If <code>TOSTOP</code> is set, background processes are inhibited from writing to their controlling terminal.

A line discipline module must record the `SUSP` suspend character and notify the Stream head when the user has typed it, and record the state of the `TOSTOP` bit and notify the Stream head when the user has changed it.

Allocation and Deallocation

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

- The Stream is acting as a terminal,
- The Stream is not already allocated as a controlling terminal, and
- The Stream is opened by a session leader that does not have a controlling terminal.

Drivers and modules can inform the Stream head to act as a terminal Stream by sending an `M_SETOPTS` message with the `SO_ISTTY` flag set upstream. This state may be changed by sending an `M_SETOPTS` message with the `SO_ISNTTY` flag set upstream.

Controlling terminals are allocated with the `open(2)` system call. A Stream head must be informed that it is acting as a terminal by an `M_SETOPTS` message sent upstream before or while the Stream is being opened by a potential controlling process. If the Stream head is opened before receiving this message, the Stream is not allocated as a controlling terminal.

Hungup Streams

When a Stream head receives an `M_HANGUP` message, it is marked as hung-up. Streams that are marked as Hungup are allowed to be reopened by their session leader if they are allocated as a controlling terminal, and by any process if they are not allocated as a controlling terminal. This way, the hangup error can be cleared without forcing all file descriptors to be closed first.

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

Hangup Signals

When the `SIGHUP` signal is generated via an `M_HANGUP` message (instead of an `M_SIG` or `M_PCSIG` message), the signal is sent to the controlling process instead of the foreground process group, since the allocation and deallocation of controlling terminals to a session is the responsibility of that process group.

Accessing the Controlling Terminal

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

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

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

The `TOSTOP` flag is set upon reception of an `M_SETOPTS` message with the `SO_TOSTOP` flag set in the *so_flags* field. It is cleared upon reception of an `M_SETOPTS` message with the `SO_TONSTOP` flag set.

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

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

Module and Driver Environment

Modules and drivers are processing elements in STREAMS. A Stream device driver is similar to a conventional device driver. It is opened like a character driver and is responsible for the system interface to the device.

STREAMS modules and drivers are structurally similar. The call interfaces to driver routines are identical to interfaces used for modules. Drivers and modules must declare `streamtab`, `qinit`, and `module_info` structures. Within the STREAMS mechanism drivers are required elements, but modules are optional.

There are three significant differences between modules and drivers:

- A driver must be able to handle interrupts from a device, so the driver will include an interrupt handler routine.
- A driver may have multiple Streams connected to it.
- Drivers exist within the file system name space; you use the system call `open` to open them. Modules don't process interrupts and can only be pushed onto an already opened Stream.

User context is not generally available to STREAMS module procedures and drivers.



Caution – STREAMS driver and module put procedures and service procedures have no user context. They cannot block.

Module and Driver Declarations

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

Code Example 7-1 Module and Driver Declarations

```
#include <sys/types.h>
#include <sys/stream.h>
#include <sys/param.h>
#include <sys/stropts.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

static struct module_info rminfo =
    { 0x08, "mod", 0, INFPSZ, 0, 0 };
static struct module_info wminfo =
    { 0x08, "mod", 0, INFPSZ, 0, 0 };
static int modopen (queue_t *, dev_t *, int, int, cred_t *);
static int modput (queue_t *, mblk_t *);
static int modclose (queue_t *, int, cred_t *);

static struct qinit rinit = {
    modput, NULL, modopen, modclose, NULL, &rminfo, NULL };
static struct qinit winit = {
    modput, 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. For information on the data structures discussed, see the man(9S) section of SunOS 5.3 Reference Manual.

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

The header files, `types.h` and `stream.h`, are always required for modules and drivers. The header file, `param.h`, contains definitions for `NULL` and other values for STREAMS modules and drivers. See also *Writing Device Drivers*.

The `streamtab(9S)` contains `qinit(9S)` values for the read and write queues. The `qinit` structures in turn point to a `module_info(9S)` and an optional `module_stat` structure. The two required structures are:

Code Example 7-2 `qinit`

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

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

The `qinit` structure contains the queue procedures: `put`, `service`, `open`, and `close`. All modules and drivers with the same `streamtab` point to the same read side and write side structure(s). The structure is meant to be software read-only, as any changes to it affect all instantiations of that module in all Streams. Pointers to the open and close procedures must be contained in the read `qinit` structure. These fields are ignored on the write-side. Our example has no `service` procedure on the read-side or write-side.

The `module_info` contains identification and limit values. All queues associated with a certain driver/module share the same `module_info` structures. The `module_info` structures define the characteristics of that driver/module's queues. As with the `qinit`, this structure is intended to be software read-only. However, the four limit values (`q_minpsz`, `q_maxpsz`, `q_hiwat`, `q_lowat`) are copied to a `queue` structure where they are modifiable via

`strqset()`. In the example, the flow control high and low water marks are zero since there is no `service` procedure 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 `/kernel/strmod/modname` file (or alternately `/usr/kernel/strmod`) used to configure the module; the prefix for `streamtab`, used in configuring the module; and the module name field in the `module_info` structure. The module name must be the same as that of `/kernel/strmod/modname` for autoconfiguration (for example `/kernel/strmod/ldterm`). Each module ID and module name should be unique in the system.

Minimum and maximum packet sizes are intended to limit the total number of characters contained in `M_DATA` messages 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-side.

The `module_stat` is optional. Currently, there is no STREAMS support for per-module statistical information gathering. For STREAMS framework statistics, use `netstat -m`.

Null Module Example

The null module procedures are as follows:

Code Example 7-3 Null Module Example

```
static int modopen(
    queue_t*q,           /* pointer to the read queue */
    dev_t*devp,          /* ptr to major/minor device # */
    int flag,            /* file flags */
    int sflag,           /* stream open flags */
    cred_t*credp)        /* ptr to a credentials struct */
{
    qprocson(q);         /* enable put/srv routines */
    return (0);          /* return success */
}

static int modput(
    queue_t*q,           /* pointer to the queue */
    ...
```

```

    mblk_t*mp)                /* message pointer */
{
    putnext(q, mp);           /* pass message through */
    return (0);
}

/* Note: we only need one put procedure that can be used
 * for both read-side and write-side.
 */

static int modclose(
    queue_t*q,                /* pointer to the read queue */
    int flag,                  /* file flags */
    cred_t*credp)              /* ptr to a credentials structure */
{
    qprocsoff(q);
    return (0);
}

```

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

modopen illustrates the open call arguments and return value. The arguments are the read queue pointer (*q*), the pointer (*devp*) to the major/minor device number, the file flags (*flag*, defined in `open(9E)`), the Stream open flag (*sflag*), and a pointer to a credentials structure (*credp*). The Stream open flag can take the following values:

Table 7-1 Stream open flag

sflag value	definition
MODOPEN	normal module open
CLONEOPEN	clone driver open
0	normal driver open

The return value from `open` is 0 for success and an error number for failure. If a driver is called with the `CLONEOPEN` flag and the driver supports the clone feature, the device number pointed to by the *devp* should be set by the driver to an unused device number accessible to that driver. This should be an entire device number (major and minor device number). The open procedure for a module is called on `I_PUSH` and on all subsequent `open` calls to the same

Stream. During a push, a nonzero return value causes the `I_PUSH` to fail and the module to be removed from the Stream. If an error is returned by a module during a push, the `ioctl` fails and the Stream remains intact.

In the next example, the module open fails if not opened by the super-user. Permission checks in module and driver open routines should be done with the `drv_priv()` routine.

```
error = drv_priv(credp);
if (error) == EPERM /* not super-user */
    return EPERM;
```

In the null module example, *modopen* enables its `put` and `srv` routines and returns successfully. *modput* illustrates the common interface to `put` procedures. The arguments are the read or write queue 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` function (see Appendix C). `putnext` calls the `put` procedure of the next queue in the proper direction.

The module close routine is only called on an `I_POP` `ioctl` or on the last `close` call of the Stream. The arguments are the read queue pointer, the file flags as in *modopen*, and a pointer to a credentials structure. The return value is 0 on success and *errno* on failure.

Module and Driver `ioctls`

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

The character I/O mechanism handles all `ioctl(2)` system calls transparently. That is, the kernel expects all `ioctls` to be handled by the device driver associated with the character special file on which the call is sent. All `ioctl` calls are sent to the driver, which is expected to perform all validation and processing other than file descriptor validity checking. The operation of any specific `ioctl` is dependent on the device driver. If the driver requires data to

be transferred in from user space, it will use the kernel `ddi_copyin()` function. It may also use `ddi_copyout()` to transfer any data results to user space.

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

First, there are a set of generic STREAMS `ioctl` command values (see `ioctl(2)`) recognized and processed by the Stream head. These are described in `streamio(7)`. The operation of the generic STREAMS `ioctls` are generally independent of the presence of any specific module or driver on the Stream.

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

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

STREAMS provides the capability for user processes to perform control functions on specific modules and drivers in a Stream with `ioctl` calls. Most `streamio(7)` `ioctl` commands go no further than the Stream head. They are fully processed there and no related messages are sent downstream. However, certain commands and all unrecognized commands cause the Stream head to create an `M_IOCTL` message which includes the `ioctl` arguments and send the message downstream to be received and processed by a specific module or driver. The `M_IOCTL` message is the initial message type which carries `ioctl` information to modules. Other message types are used to complete the `ioctl` processing in the Stream. In general, each module must uniquely recognize and act on specific `M_IOCTL` messages.

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

General ioctl Processing

STREAMS blocks a user process that issues an `ioctl` and causes the Stream head to generate an `M_IOCTL` message. The process remains blocked until one of the following occurs:

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

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

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

The form of an `M_IOCTL` message is a single `M_IOCTL` message block followed by zero or more `M_DATA` blocks (see Figure B-1 in Appendix B, “Message Types”). The `M_IOCTL` message block contains an `iocblk(9S)` structure.

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

For an `I_STR` `ioctl`, `ioc_cmd` contains the command supplied by the user in the `striocblk` structure defined in `streamio(7)`. For others, it is the value of the `cmd` argument in the call to `ioctl()`.

If a module or driver determines an `M_IOCTL` message is in error for any reason, it must produce the negative acknowledgment message. This is done by setting the message type to `M_IOCNAK` and sending the message upstream. No data or a return value can be sent to a user in this case. If `ioc_error` is set to 0, the Stream head will cause the `ioctl` call to fail with `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 nonzero value in both `M_IOCACK` and `M_IOCNAK`. This will cause that value to be returned as an error number to the process that sent the `ioctl`.

If a module looks at what `ioctls` of other modules are doing, the module should not search for a specific `M_IOCTL` on the write-side but look for `M_IOCACK` or `M_IOCNAK` on the read-side. For example, the module sees `TCSETA` (see `termio(7)`) failing and searches for what is being set. The module should look at it and save away the answer but not use it. The read-side processing knows that the module is waiting for an answer for the `ioctl`. When the read-side processing sees an “ack” or “nak” next time, it checks if it is the same `ioctl` (here `TCSETA`) and if it is, the module may use the answer previously saved.

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

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

I_STR ioctl Processing

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

If the module is looking at, for example, the `TCSETA/TCGETA` group of `ioctl` calls as they pass up or down a Stream, it must never assume that because `TCSETA` comes down that it actually has a data buffer attached to it. The user may have formed `TCSETA` as an `I_STR` call and accidentally given a null data buffer pointer. One must always check `b_cont` to see if it is `NULL` before using it as an index to the data block that goes with `M_IOCTL` messages.

The `TCGETA` call, if formed as an `I_STR` call with a data buffer pointer set to a value by the user, will always have a data buffer attached to `b_cont` from the main message block. If one assumes that the data block is not there and allocates a new buffer and assigns `b_cont` to point at it, the original buffer will be lost. Thus, before assuming that the `ioctl` message does not have a buffer attached, one should check first.

The following example, Code Example 7-4, illustrates processing associated with an `I_STR ioctl`. `lpdoioctl` is called to process trapped `M_IOCTL` messages:

Code Example 7-4 `I_STR ioctl`

```
static void
lpdoioctl(
    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
 * (for this example)
 */
if (iocp->ioc_count != sizeof(short))
    goto iocnak;
if (mp->b_cont == NULL)
    goto lognak; /* not shown in this example */
/* Actual data is in 2nd message block */
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 driver `M_IOCTL` processing which also applies to modules. However, at case *default*, a module would not “nak” an unrecognized command, but would pass the message on. 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. 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 nonzero, the Stream head would copy that many bytes from the second - Nth message blocks into the user buffer. You must set *ioc* count if you want to pass any data back to the user.

Transparent ioctl Processing

The transparent STREAMS `ioctl` mechanism allows application programs to perform module and driver control functions with `ioctls` other than `I_STR`. It is intended to transparently support applications developed prior to the introduction of STREAMS. It alleviates the need to recode and recompile the

user level software to run over STREAMS files. More importantly, it relieves applications of the burden of packaging their `ioctl` requests into the form demanded by `I_STR`.

The mechanism extends the data transfer capability for STREAMS `ioctl` calls beyond that provided in the `I_STR` form. Modules and drivers can transfer data between their kernel space and user space in any `ioctl` which has a value of the `command` argument not defined in `streamio(7)`. These `ioctl`s are known as transparent `ioctl`s to differentiate them from the `I_STR` form. Transparent processing support is necessary when existing user level applications perform `ioctl`s on a non-STREAMS character device and the device driver is converted to STREAMS. The `ioctl` data can be in any format mutually understood by the user application and module.

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

This mechanism is needed because user context does not exist in modules and drivers when `ioctl` processing occurs. This prevents them from using the kernel `ddi_copyin()/ddi_copyout()` functions. For example, consider the following `ioctl` call:

```
ioctl (stream_filedes, user_command, &ioctl_struct);
```

where `ioctl_struct` is a structure whose members are:

```
struct ioctl_struct {
    int      stringlen;
    char     *string;
    struct other_struct*other1;
};
```

To read (or write) the elements of `ioctl_struct`, a module would have to cause a series of `ddi_copyin()/ddi_copyout()` calls at the stream head, using pointer information from a prior `ddi_copyin()` to transfer additional data. A non-STREAMS character driver could directly execute these copy functions because user context exists during all system calls to the driver. However, in STREAMS, user context is only available to modules and drivers in their open and close routines.

The transparent mechanism enables modules and drivers to request that the Stream head perform a `ddi_copyin()` or `ddi_copyout()` on their behalf to transfer `ioctl` data between their kernel space and various user space locations. The related data is sent in message pairs exchanged between the Stream head and the module. A pair of messages is required so that each transfer can be acknowledged. In addition to `M_IOCTL`, `M_IOCACK`, and `M_IOCNAK` messages, the transparent mechanism also uses `M_COPYIN`, `M_COPYOUT`, and `M_IOCDATA` messages.

The general processing by which a module or a driver reads data from user space for the transparent case involves pairs of request/response messages, as follows:

1. The Stream head does not recognize the *command* argument of an `ioctl` call and creates a transparent `M_IOCTL` message (the `iocblk` structure has a `TRANSPARENT` indicator, see “*Transparent ioctl Messages*”) containing the value of the *arg* argument in the call. It sends the `M_IOCTL` message downstream.
2. A module receives the `M_IOCTL` message, recognizes the *ioc_cmd*, and determines that it is `TRANSPARENT`.
3. If the module requires user data, it creates an `M_COPYIN` message to request a `copyin()` of user data. The message will contain the address of user data to copy in and how much data to transfer. It sends the message upstream.
4. The Stream head receives the `M_COPYIN` message and uses the contents to `copyin()` the data from user space into an `M_IOCDATA` response message that it sends downstream. The message also contains an indicator of whether the data transfer succeeded.
5. The module receives the `M_IOCDATA` message and processes its contents.

The module may use the message contents to generate another `M_COPYIN`. Steps 3 through 5 may be repeated until the module has requested and received all the user data to be transferred.

6. When the module completes its data transfer, it performs the `ioctl` processing and sends an `M_IOCACK` message upstream to notify the Stream head that `ioctl` processing has successfully completed.

Writing data from a module to user space is similar except that the module uses an `M_COPYOUT` message to request the Stream head to write data into user space. In addition to length and user address, the message includes the data to be copied out. In this case, the `M_IOCADATA` response will not contain user data, only an indication of success or failure.

The module may mix `M_COPYIN` and `M_COPYOUT` messages in any order. However, each message must be sent one at a time; the module must receive the associated `M_IOCADATA` response before any subsequent `M_COPYIN`/`M_COPYOUT` request or “ack/nak” message is sent upstream. After the last `M_COPYIN`/`M_COPYOUT` message, the module must send an `M_IOCACK` message (or `M_IOCNAK` in the event of a detected error condition).



Caution – For a transparent `M_IOCTL`, user data can not be returned with an `M_IOCACK` message. The data must have been sent with a preceding `M_COPYOUT` message.

Transparent ioctl Messages

The form of the `M_IOCTL` message generated by the Stream head for a transparent `ioctl` is a single `M_IOCTL` message block followed by one `M_DATA` block. The form of the `iocblk` structure in the `M_IOCTL` block is the same as described under “General ioctl Processing”. However, `ioc_cmd` is set to the value of the *command* argument in the `ioctl` system call and `ioc_count` is set to `TRANSPARENT`. `TRANSPARENT` distinguishes the case where an `I_STR` `ioctl` may specify a value of `ioc_cmd` equivalent to the *command* argument of a transparent `ioctl`. The `M_DATA` block of the message contains the value of the *arg* parameter in the call.



Caution – Modules that process a specific *ioc_cmd* which did not validate the *ioc_count* field of the `M_IOCTL` message will break if transparent `ioctls` with the same command are performed from user space.

`M_COPYIN`, `M_COPYOUT`, and `M_IOCADATA` messages and their use are described in more detail in Appendix B, “Message Types”.

Transparent ioctl Examples

Following are three examples of transparent `ioctl` processing. The first illustrates `M_COPYIN`. The second illustrates `M_COPYOUT`. The third is a more complex example showing state transitions combining both `M_COPYIN` and `M_COPYOUT`.

M_COPYIN Example

In this example, the contents of a user buffer are to be transferred into the kernel as part of an `ioctl` call of the form

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

where `bufadd` is a structure of type *struct address* whose elements are:

```
struct address {
    intad_len;                /* buffer length in bytes */
    caddr_tad_addr;           /* buffer address */
};
```

This requires two pairs of messages (request/response) following receipt of the `M_IOCTL` message. The first will `copyin` the structure and the second will `copyin` the buffer. This example illustrates processing that supports only the transparent form of `ioctl`. `xxwput` is the write-side `put` procedure for module or driver `xx`:

```
struct address {
    int    ad_len;            /* same members as in user space */
    caddr_t ad_addr;          /* length in bytes */
};
/* state values (overloaded in private field) */
#define GETSTRUCT 0           /* address structure */
#define GETADDR 1             /* byte string from ad_addr */

static void xxioc(queue_t *q, mblk_t *mp);

static int
xxwput(q, mp)
    queue_t *q;                /* write queue */
    mblk_t *mp;
{
```

```

struct iocblk *iocbp;
struct copyreq *cqp;

switch (mp->b_datap->db_type) {
    .
    .
    .
case M_IOCTL:
    iocbp = (struct iocblk *)mp->b_rptra;
    switch (iocbp->ioc_cmd) {
        /* do non-transparent processing.
        */

        /* Reuse M_IOCTL block for M_COPYIN request */
case SET_ADDR:
    cqp = (struct copyreq *)mp->b_rptra;

    /* Get user space structure address from
    * linked M_DATA block */

    cqp->cq_addr = (caddr_t) *(long *)mp->b_cont->b_rptra;
    freemsg(mp->b_cont); /* MUST free linked blks */
    mp->b_cont = NULL;
    /* to identify response */
    cqp->cq_private = (mblock_t *)GETSTRUCT;

    /* Finish describing M_COPYIN message */

    cqp->cq_size = sizeof(struct address);
    cqp->cq_flag = 0;
    mp->b_datap->db_type = M_COPYIN;
    mp->b_wptra=mp->b_rptra+sizeof(struct copyreq);
    qreply(q, mp);
    break;

default: /* M_IOCTL not for us */
    /* if module, pass on */
    /* if driver, nak ioctl */
    break;
} /* switch (iocbp->ioc_cmd) */
break;
case M_IOCDATA:
    /* all M_IOCDATA processing done here */

```

```

        xxioc(q, mp);
        break;
    }
    return (0);
}

```

xxwput verifies that the `SET_ADDR` is TRANSPARENT to avoid confusion with an `I_STR` `ioctl`, which uses a value of *ioc_cmd* equivalent to the command argument of a transparent `ioctl`. When sending an `M_IOCNAK`, freeing the linked `M_DATA` block is not mandatory as the Stream head will free it. However, this returns the block to the buffer pool more quickly.

In this and all following examples in this section, the message blocks are reused to avoid the overhead of releasing and allocating, this is standard practice.

Note – The Stream head will guarantee that the size of the message block containing an `iocblk` structure will be large enough also to hold the `copyreq` and `copyresp` structures.

cq_private is set to contain state information for `ioctl` processing (this identifies what the subsequent `M_IOCTLDATA` response message contains). Keeping the state in the message makes the message self-describing and simplifies the `ioctl` processing. `M_IOCTLDATA` processing is done in *xxioc*. Two `M_IOCTLDATA` types are processed, *GETSTRUCT* and *GETADDR*:

```

xxioc(queue_t *q, mblk_t *mp)          /* M_IOCTLDATA processing */
{
    struct iocblk *iocbp;
    struct copyreq *cqp;
    struct copyresp *csp;
    struct address *ap;

    csp = (struct copyresp *)mp->b_rptr;
    iocbp = (struct iocblk *)mp->b_rptr;

    /* validate this M_IOCTLDATA is for this module */
    switch (csp->cp_cmd) {

    case SET_ADDR:
        if (csp->cp_rval){ /*GETSTRUCT or GETADDRfail*/
            freemsg(mp);

```

```

        return;
    }
    switch ((int)csp->cp_private){ /*determine state*/

case GETSTRUCT:    /* user structure has arrived */
    /* reuse M_IOCTLDATA block */
    mp->b_datap->db_type = M_COPYIN;
    cqp = (struct copyreq *)mp->b_rptra;
    /* user structure */
    ap = (struct address *)mp->b_cont->b_rptra;
    /* buffer length */
    cqp->cq_size = ap->ad_len;
    /* user space buffer address */
    cqp->cq_addr = ap->ad_addr;
    freemsg(mp->b_cont);
    mp->b_cont = NULL;
    cqp->cq_flag = 0;
    csp->cp_private=(mblk_t *)GETADDR; /*nxt st*/
    greply(q, mp);
    break;

case GETADDR:      /* user address is here */
    /* hypothetical routine */
    if (xx_set_addr(mp->b_cont) == FAILURE) {
        mp->b_datap->db_type = M_IOCTLNAK;
        iocbp->ioc_error = EIO;
    } else {
        mp->b_datap->db_type=M_IOCTLACK; /*success*/
        /* may have been overwritten */
        iocbp->ioc_error = 0;
        iocbp->ioc_count = 0;
        iocbp->ioc_rval = 0;
    }
    mp->b_wptra=mp->b_rptra + sizeof (struct iocblk);
    freemsg(mp->b_cont);
    mp->b_cont = NULL;
    greply(q, mp);
    break;
default: /* invalid state: can't happen */
    freemsg(mp->b_cont);
    mp->b_cont = NULL;
    mp->b_datap->db_type = M_IOCTLNAK;
    mp->b_wptra = mp->rptra + sizeof(struct iocblk);
    /* may have been overwritten */

```

```

        iocbp->ioc_error = EINVAL;
        greply(q, mp);
        break;
    }
    break; /* switch (cp_private) */

default: /* M_IOCTLDATA not for us */
    /* if module, pass message on */
    /* if driver, free message */
    break;
} /* switch (cp_cmd) */
}

```

`xx_set_addr` is a routine (not shown in the example) that processes the user address from the `ioctl`. Since the message block has been reused, the fields that the Stream head will examine (denoted by “may have been overwritten”) must be cleared before sending an `M_IOCNAK`.

M_COPYOUT Example

In this example, the user wants option values for this Stream device to be placed into the user’s *options* structure (see beginning of example code, below). This can be accomplished by use of a transparent `ioctl` call of the form

```
ioctl(fd, GET_OPTIONS, (caddr_t) &optadd)
```

or, alternately, by use of a `I_STR` call

```
ioctl(fd, I_STR, (caddr_t) &opts_striocctl)
```

In the first case, *optadd* is declared *struct options*. In the `I_STR` case, *opts_striocctl* is declared *struct striocctl* where *opts_striocctl.ic_dp* points to the user *options* structure.

This example illustrates support of both the `I_STR` and transparent forms of an `ioctl`. The transparent form requires a single `M_COPYOUT` message following receipt of the `M_IOCTL` to copyout the contents of the structure. `xxwput` is the write-side put procedure for module or driver `xx`:

```

struct options {                                /* same members as in user space */
    int      op_one;
    int      op_two;
    short    op_three;
    long     op_four;
};

static int
xxwput(
    queue_t *q,                                /* write queue */
    mblk_t *mp)
{
    struct iocblk *iocbp;
    struct copyreq *cqp;
    struct copyresp *csp;
    int transparent = 0;

    switch (mp->b_datap->db_type) {
        .
        .
        .
    case M_IOCTL:
        iocbp = (struct iocblk *)mp->b_rptr;
        switch (iocbp->ioc_cmd) {

            case GET_OPTIONS:
                if (iocbp->ioc_count == TRANSPARENT) {
                    transparent = 1;
                    cqp = (struct copyreq *)mp->b_rptr;
                    cqp->cq_size = sizeof(struct options);
                    /* Get struct address from linked M_DATA block */
                    cqp->cq_addr = (caddr_t)
                        *(long *)mp->b_cont->b_rptr;
                    cqp->cq_flag = 0;

                    /* No state necessary - we will only ever
                     * get one M_IOCTL from the Stream head
                     * indicating success or failure for
                     * the copyout */
                }
            }
        }
    }
}

```

```

        if (mp->b_cont)
            freemsg(mp->b_cont); /*over written below*/
        if ((mp->b_cont=alloca(sizeof(struct options),
            BPRI_MED)) == NULL) {
            mp->b_datap->db_type = M_IOCNAK;
            iocbp->ioc_error = EAGAIN;
            greply(q, mp);
            break;
        }
        /* hypothetical routine */
        xx_get_options(mp->b_cont);
        if (transparent) {
            mp->b_datap->db_type = M_COPYOUT;
            mp->b_wptr = mp->b_rptr +
                sizeof(struct copyreq);
        } else {
            mp->b_datap->db_type = M_IOCACK;
            iocbp->ioc_count = sizeof(struct options);
        }
        greply(q, mp);
        break;

    default: /* M_IOCTL not for us */
        /*if module, pass on;if driver, nak ioctl*/

        break;
    } /* switch (iocbp->ioc_cmd) */
    break;

case M_IOCDATA:
    csp = (struct copyresp *)mp->b_rptr;
    /* M_IOCDATA not for us */
    if (csp->cmd != GET_OPTIONS) {
        /*if module/pass on, if driver/free message*/

        break;
    }
    if ( csp->cp_rval ) {
        freemsg(mp); /* failure */
        return (0);
    }
    /* Data successfully copied out, ack */

    /* reuse M_IOCDATA for ack */
    mp->b_datap->db_type = M_IOCACK;

```

```

        mp->b_wptr = mp->b_rptr + sizeof(struct iocblk);
        /* may have been overwritten */
        iocbp->ioc_error = 0;
        iocbp->ioc_count = 0;
        iocbp->ioc_rval = 0;
        greply(q, mp);
        break;
        .
        .
        .
    } /* switch (mp->b_datap->db_type) */
    return (0);

```

Bidirectional Transfer Example

This example illustrates bidirectional data transfer between the kernel and user space during transparent `ioctl` processing. It also shows how more complex state information can be used.

The user wants to send and receive data from user buffers as part of a transparent `ioctl` call of the form

```
ioctl(fd, XX_IOCTL, (caddr_t) &addr_xxdata)
```

The user `addr_xxdata` structure defining the buffers is declared as `struct xxdata`, shown below. This requires three pairs of messages following receipt of the `M_IOCTL` message: the first to `copyin` the structure; the second to `copyin` one user buffer; and the last to `copyout` the second user buffer. `xxwput` is the write-side `put` procedure for module or driver `xx`:

```

struct xxdata {
    int                x_inlen; /* number of bytes copied in */
    caddr_t            x_inaddr; /* buf addr of data copied in */
    int                x_outlen; /* number of bytes copied out */
    caddr_t            x_outaddr; /* buf addr of data copied out */
};
/* State information for ioctl processing */
struct state {
    int                st_state; /* see below */
    struct xxdata      st_data; /* see above */
};
/* state values */

#define GETSTRUCT      0 /* get xxdata structure */

```



```

#define GETINDATA          1 /*get data from x_inaddr */
#define PUTOUTDATA         2 /* get response from M_COPYOUT */

static void xxioc(queue_t *q, mblk_t *mp);

static int
xxwput(
    queue_t *q,                      /* write queue */
    mblk_t *mp)
{
    struct iocblk *iocbp;
    struct copyreq *cqp;
    struct state *stp;
    mblk_t *tmp;

    switch (mp->b_datap->db_type) {
        .
        .
        .
    case M_IOCTL:
        iocbp = (struct iocblk *)mp->b_rptr;
        switch (iocbp->ioc_cmd) {
            case XX_IOCTL:
                /* do non-transparent processing. (See I_STR ioctl
                 * processing discussed in previous section.)
                 */
                /*Reuse M_IOCTL block for M_COPYIN request*/

                cqp = (struct copyreq *)mp->b_rptr;

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

                cqp->cq_addr = (caddr_t)
                    *(long *)mp->b_cont->b_rptr;
                freemsg(mp->b_cont);
                mp->b_cont = NULL;

                /* Allocate state buffer */

                if ((tmp = allocb(sizeof(struct state),
                    BPRI_MED)) == NULL) {
                    mp->b_datap->db_type = M_IOCNAK;
                    iocbp->ioc_error = EAGAIN;
                    greply(q, mp);
                }
            }
        }
    }
}

```

```

        break;
    }
    tmp->b_wptr += sizeof(struct state);
    stp = (struct state *)tmp->b_rptr;
    stp->st_state = GETSTRUCT;
    cqp->cq_private = tmp;

    /* Finish describing M_COPYIN message */

    cqp->cq_size = sizeof(struct xxdata);
    cqp->cq_flag = 0;
    mp->b_datap->db_type = M_COPYIN;
    mp->b_wptr=mp->b_rptr+sizeof(struct copyreq);
    greply(q, mp);
    break;

default: /* M_IOCTL not for us */
    /* if module, pass on */
    /* if driver, nak ioctl */
    break;

} /* switch (iocbp->ioc_cmd) */
break;

case M_IOCTLDATA:
    xxioc(q, mp); /*all M_IOCTLDATA processing here*/
    break;
    .
    .
    .
} /* switch (mp->b_datap->db_type) */
}

```

xxwput allocates a message block to contain the state structure and reuses the *M_IOCTL* to create an *M_COPYIN* message to read in the *xxdata* structure.

M_IOCTLDATA processing is done in *xxioc*:

```

xxioc(                                     /* M_IOCTLDATA processing */
    queue_t *q,
    mblk_t *mp)
{
    struct iocblk *iocbp;
    struct copyreq *cqp;

```

```

struct copyresp *csp;
struct state *stp;
mblk_t *xx_indata();

csp = (struct copyresp *)mp->b_rptr;
iocbp = (struct iocblk *)mp->b_rptr;
switch (csp->cp_cmd) {

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

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

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

case GETINDATA: /* data successfully copied in */
    /* Process input, return output */
    if ((mp->b_cont = xx_indata(mp->b_cont))
        == NULL) { /* hypothetical */
        /* fail xx_indata */
        mp->b_datap->db_type = M_IOCNAK;
        mp->b_wptr = mp->b_rptr +
            sizeof(struct iocblk);
        iocbp->ioc_error = EIO;
    }
}
}

```

```

        greply(q, mp);
        break;
    }
    mp->b_datap->db_type = M_COPYOUT;
    cqp = (struct copyreq *)mp->b_rptra;
    cqp->cq_size = min(msgdsize(mp->b_cont),
        stp->st_data.x_outlen);
    cqp->cq_addr = stp->st_data.x_outaddr;
    cqp->cq_flag = 0;
    stp->st_state = PUTOUTDATA; /* next state */
    greply(q, mp);
    break;
case PUTOUTDATA: /* data copied out, ack ioctl */
    freemsg(csp->cp_private); /*state structure*/
    mp->b_datap->db_type = M_IOCACK;
    mp->b_wtptr=mp->b_rptra + sizeof (struct iocblk);
    /* may have been overwritten */
    iocbp->ioc_error = 0;
    iocbp->ioc_count = 0;
    iocbp->ioc_rval = 0;
    greply(q, mp);
    break;

default: /* invalid state: can't happen */
    freemsg(mp->b_cont);
    mp->b_cont = NULL;
    mp->b_datap->db_type = M_IOCNAK;
    mp->b_wtptr=mp->b_rptra + sizeof (struct iocblk);
    iocbp->ioc_error = EINVAL;
    greply(q, mp);
    break;
} /* switch (stp->st_state) */
break;
default: /* M_IOCdata not for us */
    /* if module, pass message on */
    /* if driver, free message */
    break;
} /* switch (csp->cp_cmd) */
}

```

At case GETSTRUCT, the user *xxdata* structure is copied into the module's state structure (pointed at by *cp_private* in the message) and the M_IOCdata message is reused to create a second M_COPYIN message to read the user data. At case GETINDATA, the input user data is processed by the *xx_indata* routine

(not supplied in the example), which frees the linked `M_DATA` block and returns the output data message block. The `M_IOCDATA` message is reused to create an `M_COPYOUT` message to write the user data. At case `PUTOUTDATA`, the message block containing the state structure is freed and an acknowledgment is sent upstream.

Care must be taken at the “can’t happen” *default* case since the message block containing the state structure (*cp_private*) is not returned to the pool because it might not be valid. This might result in a lost block. The `ASSERT` will help find errors in the module if a “can’t happen” condition occurs.

I_LIST ioctl

The `ioctl I_LIST` supports the `strconf` and `strchg` commands (see `strchg(1)`) that are used to query or change the configuration of a Stream. Only the super-user or an owner of a STREAMS device may alter the configuration of that Stream.

The `strchg` command does the following:

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

The `strconf` command does the following:

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

The `ioctl I_LIST` performs two functions. When the third argument of the `ioctl` call is set to `NULL`, the return value of the call indicates the number of modules, including the driver, present on the Stream. For example, if there are two modules above the driver, 3 is returned. On failure, *errno* may be set to a value specified in `streamio(7)`. The second function of the `I_LIST ioctl` is

to copy the module names found on the Stream to the user supplied buffer. The address of the buffer in user space and the size of the buffer are passed to the `ioctl` through a structure `str_list` that is defined as:

```
struct str_mlist {
    char l_name[FMNAMESZ+1]; /*space for holding a module name*/
};
struct str_list {
    int sl_nmods; /*#of modules for which space is allocated*/
    struct str_mlist *sl_modlist; /*addr of buf for names*/
};
```

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

The user can find out how much space to allocate by first calling the `ioctl` `I_LIST` with *arg* set to `NULL`. The `I_LIST` call with *arg* pointing to the `str_list` structure returns the number of entries that have been filled into the *sl_modlist* array (the number includes the number of modules including the driver). If there is not enough space in the *sl_modlist* array (see note) or *sl_nmods* is less than 1, the `I_LIST` call will fail and *errno* is set to `EINVAL`. If *arg* or the *sl_modlist* array points outside the allocated address space, `EFAULT` is returned.

Note – It is possible that another module was pushed on the Stream after the user invoked the `I_LIST` `ioctl` with the `NULL` argument and before the `I_LIST` `ioctl` with the structure argument was invoked.

Flush Handling

All modules and drivers are expected to handle `M_FLUSH` messages. An `M_FLUSH` message can originate at the Stream head or from a module or a driver. The first byte of the `M_FLUSH` message is an option flag that can have following values:

`FLUSHR` Flush read queue.

`FLUSHW` Flush write queue.

`FLUSHRW` Flush both, read and write, queues.

`FLUSHBAND` Flush a specified priority band only.

The next two figures further demonstrate flushing the entire Stream due to a line break. Figure 7-1 shows the flushing of the write-side of a Stream, and Figure 7-2 shows the flushing of the read-side of a Stream. In the figures dotted boxes indicate flushed queues.

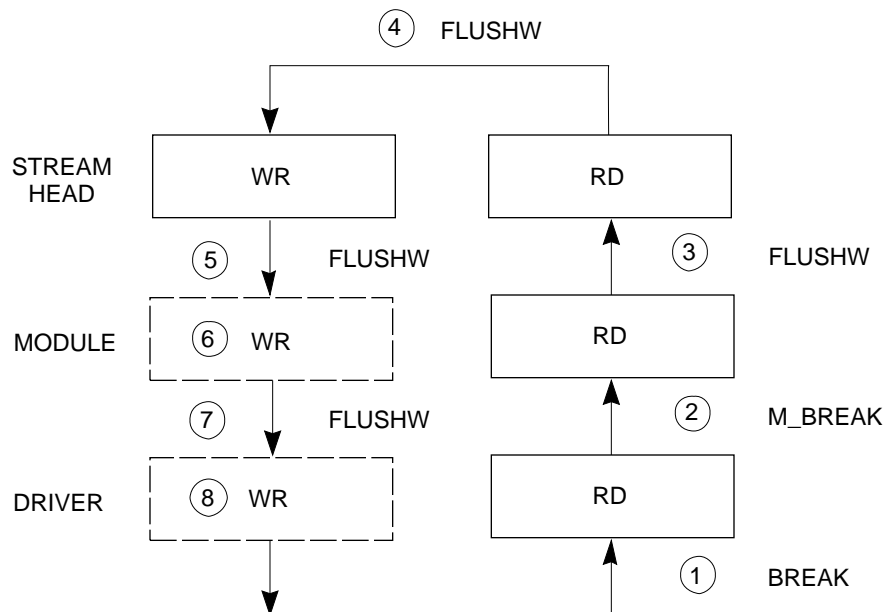


Figure 7-1 Flushing The Write-Side of A Stream

The following takes place (dotted lines mean flushed queues):

1. A break is detected by a driver.
2. The driver generates an `M_BREAK` message and sends it upstream.
3. The module translates the `M_BREAK` into an `M_FLUSH` message with `FLUSHW` set and sends it upstream.
4. The Stream head does *not* flush the write queue (no messages are ever queued there).
5. The Stream head turns the message around (sends it down the write-side).
6. The module flushes its write queue.
7. The message is passed downstream.
8. The driver flushes its write queue and frees the message.

This figure shows flushing read-side of a Stream.

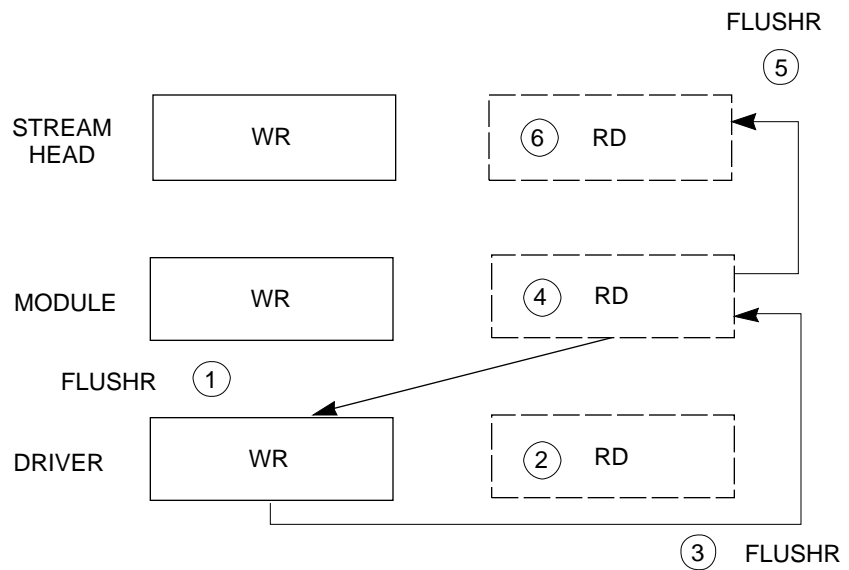


Figure 7-2 Flushing The Read-Side of A Stream

The events taking place are:

1. After generating the first `M_FLUSH` message, the module generates an `M_FLUSH` with `FLUSHR` set and sends it downstream.
2. The driver flushes its read queue.
3. The driver turns the message around (sends it up the read-side).
4. The module flushes its read queue.
5. The message is passed upstream.
6. The Stream head flushes the read queue and frees the message.

The following example shows line discipline module flush handling:

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

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

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

            putnext(q, mp);           /* pass it on */
            return(0);

    }
}
```

The Stream head turns around the `M_FLUSH` message if `FLUSHW` is set (`FLUSHR` will be cleared). A driver turns around `M_FLUSH` if `FLUSHR` is set (should mask off `FLUSHW`).

Flushing Priority Bands

The `flushband()` routine (see Appendix C, “STREAMS Utilities”) provides the module and driver with the capability to flush messages associated with a given priority band. A user can flush a particular band of messages by issuing:

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

where *bandp* is a pointer to a structure `bandinfo` that has a format:

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

The *bi_flag* field may be one of `FLUSHR`, `FLUSHW`, or `FLUSHRW`.

The following example shows flushing according to the priority band:

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

case M_FLUSH:
    if (*bp->b_rptr & FLUSHBAND) {
        if (*bp->b_rptr & FLUSHW)
            flushband(wrq, FLUSHDATA, *(bp->b_rptr + 1));
        if (*bp->b_rptr & FLUSHR)
            flushband(rdq, FLUSHDATA, *(bp->b_rptr + 1));
    } else {
        if (*bp->b_rptr & FLUSHW)
            flushq(wrq, FLUSHDATA);
        if (*bp->b_rptr & FLUSHR)
            flushq(rdq, FLUSHDATA);
    }
    /*
     * modules pass the message on;
     * drivers shut off FLUSHW and loop the message
     * up the read-side if FLUSHR is set; otherwise,
     * drivers free the message.
     */
    break;
```

Note that modules and drivers are not required to treat messages as flowing in separate bands. Modules and drivers can view the queue having only two bands of flow, normal and high priority. However, the latter alternative will flush the entire queue whenever an M_FLUSH message is received.

One use of the field *b_flag* of the `msgb` structure is provided to give the Stream head a way to stop M_FLUSH messages from being reflected forever when the Stream is being used as a pipe. When the Stream head receives an M_FLUSH message, it sets the MSGNOLoop flag in the *b_flag* field before reflecting the message down the write-side of the Stream. If the Stream head receives an M_FLUSH message with this flag set, the message is freed rather than reflected.

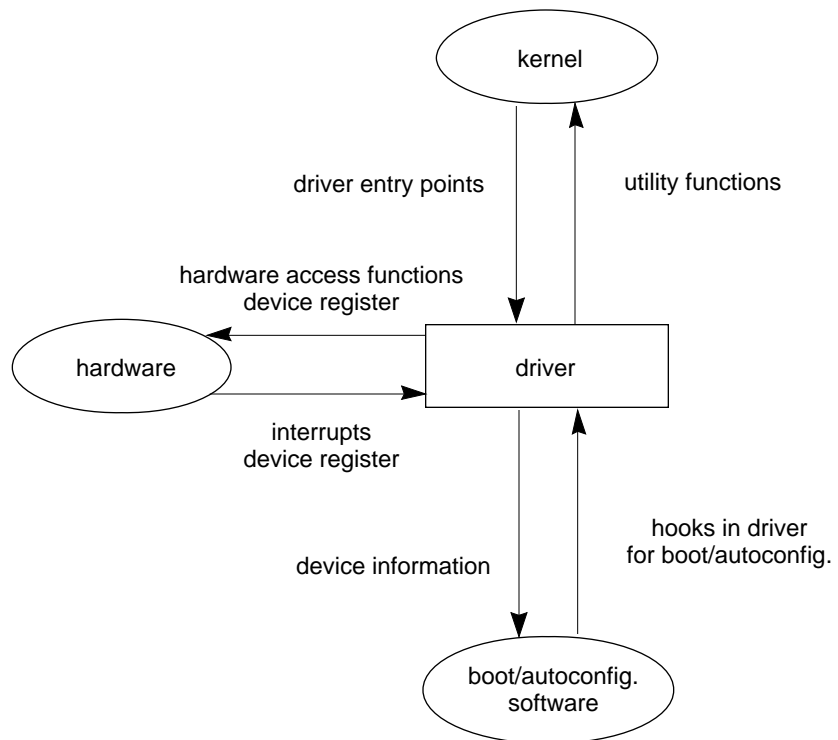


Figure 7-3 Interfaces Affecting Drivers

The set of STREAMS utilities available to drivers are listed in Appendix C. No system-defined macros that manipulate global kernel data or introduce structure-size dependencies are permitted in these utilities. Therefore, some utilities that have been implemented as *macros* in the prior Solaris system releases are implemented as *functions* in SunOS 5.x. This does not preclude the existence of both *macro* and *function* versions of these utilities. It is intended that driver source code will include a header file that picks up *function* declarations while the core operating system source includes a header file that defines the *macros*. With the DKI interface the following STREAMS utilities are implemented as C programming language functions: `datamsg`, `OTHERQ`, `putnext`, `RD`, and `WR`.

Replacing *macros* such as `RD()` with *function* equivalents in the driver source code allows driver objects to be insulated from changes in the data structures and their size, further increasing the useful lifetime of driver source code and objects. Multithreaded drivers are also protected against changes in implementation-specific STREAMS synchronization.

The DKI interface defines an interface suitable for drivers and there is no need for drivers to access global kernel data structures directly. The kernel functions `drv_getparm` and `drv_setparm` are provided for reading and writing information in these structures. This restriction has an important consequence. Since drivers are not permitted to access global kernel data structures directly, changes in the contents/offsets of information within these structures will not break objects. The `drv_getparm(9f)` and `drv_setparm(9f)` functions are described in more detail in the appropriate sections of the *man Pages(9F): DDI and DKI Kernel Functions Manual*.

Device Driver Interface and Driver–Kernel Interface

The Device Driver Interface (DDI) is a SunOS 5.3 interface that facilitates driver portability across different Solaris versions on the SPARC hardware. The Driver–Kernel Interface (DKI) is an interface that also facilitates driver source code portability across implementations of SVR4 on all machines. DKI driver code, however, will have to be recompiled on the machine on which it is to run.

The most important distinction between the DDI and the DKI lies in scope. The DDI addresses vendor specific architecture interfaces (see note below) for block, character, and STREAMS interface drivers and modules. For more information see *Writing Device Drivers*.

STREAMS Interface

The entry points from the kernel into STREAMS drivers and modules are through the `qinit` structures (see Appendix A, “STREAMS Data Structures”) pointed to by the `streamtab` structure, `prefixinfo`. STREAMS drivers may need to define additional entry points to support the interface with boot/autoconfiguration software and the hardware (for example, an interrupt handler).

Here is a simple incomplete example of a driver header. For the complete version see Appendix E, “Configuration”, which has both data structures and entry points. If the STREAMS module has prefix *mod* then the declaration is of the form:

```
static int modrput(queue_t*, mblk_t*);
static int modrsrv(queue_t*);
static int modopen(queue_t*, dev_t*, int, int, cred_t*);
static int modclose(queue_t*, int, cred_t*);

static int modwput(queue_t*, mblk_t*);
static int modwsrv(queue_t*);

static struct qinit rdinit =
    {modrput, modrsrv, modopen, modclose, NULL, NULL, NULL};
static struct qinit wrinit =
    {modwput, modwsrv, NULL, NULL, NULL, NULL, NULL};
struct streamtab modinfo = {&rdinit, &wrinit, NULL, NULL};
```

where

- *modrput* is the module’s read queue put procedure
- *modrsrv* is the module’s read queue service procedure
- *modopen* is the open routine for the module
- *modclose* is the close routine for the module
- *modwput* is the put procedure for the module’s write queue, and
- *modwsrv* is the service procedure for the module’s write queue

Each `qinit` structure can point to four entry points. (An additional function pointer has been reserved for future use and must not be used by drivers or modules.) These four function pointer fields in the `qinit` structure are: *qi_putp*, *qi_srvp*, *qi_qopen*, and *qi_close*.

The utility functions that can be called by STREAMS drivers and modules are listed in Appendix C. They must follow the call and return syntaxes specified in the appendix. Manual pages relating to the Driver–Kernel Interface and Device Driver Interface are provided in *man Pages(9F): DDI and DKI Kernel Functions* the System for STREAMS Drivers and Modules.

Configuring the System for STREAMS Drivers and Modules

To configure the system to use your driver or module, you must use a number of kernel interfaces. These consist of making it a kernel loadable module.

For a more in depth discussion of this information, please refer to Appendix E, “Configuration” and the examples there.

Design Guidelines

This section summarizes guidelines common to the design of STREAMS modules and drivers. See Chapter 8, “Modules” and Chapter 9, “Drivers” for additional rules pertaining to modules and drivers.

Rules for Modules and Drivers

Below are some rules for Modules and Drivers

1. Modules and drivers are not associated with any process, and therefore have no concept of process or user context, except during open and close routines (see “*Rules for Open/Close Routines*”).
2. Every module and driver must process an `M_FLUSH` message according to the value of the argument passed in the message.
3. A module or a driver should not change the contents of a data block whose reference count is greater than 1 (see `dupmsg()` in Appendix C) because other modules/drivers that have references to the block may not want the data changed. To avoid problems, data should be copied to a new block and then changed in the new one.
4. Modules and drivers should manipulate queues and manage buffers only with the routines provided for that purpose, (see Appendix C).

5. Modules and drivers should not require the data in an `M_DATA` message to follow a particular format, such as a specific alignment.
6. Care must be taken when modules are mixed and matched, because one module may place different semantics on the priority bands than another module. The specific use of each band by a module should be included in the service interface specification.

When designing modules and drivers that make use of priority bands one should keep in mind that priority bands merely provide a way to impose an ordering of messages on a queue. The priority band is not used to determine the service primitive. Instead, the service interface should rely on the data contained in the message to determine the service primitive.

7. Drivers must `NAK` all unrecognized `M_IOCTL` messages.
8. Drivers must silently discard unrecognized message types.
9. Modules must forward all unrecognized message types.

Rules for Open/Close Routines

Here are some rules for Open/Close Routines

1. `open` and `close` routines must use condition variables to access the functionality that was provided before by `sleep`.
2. The `open` routine should return zero on success or an error number on failure. If the `open` routine is called with the `CLONEOPEN` flag, the device number should be set by the driver to an unused device number accessible to that driver. This should be an entire device number (major/minor).
3. `open` and `close` routines have user context.
4. If a module or a driver wants to allocate a controlling terminal, it should send an `M_SETOPTS` message to the Stream head with the `SO_ISTTY` flag set. Otherwise signaling will not work on the Stream.
5. A driver or module must call `qprocson` to enable its `put` and service routines and `qprocoff` to disable them.

Rules for `ioctl`s

Here are some rules for `ioctl`s

- Do not change the `ioc_id`, `ioc_uid`, `ioc_gid`, or `ioc_cmd` fields in an `M_IOCTL` message.
- These rules also apply to fields in an `M_IOCTLDATA`, `M_COPYIN`, and `M_COPYOUT` message. (Field names are different; see Appendix A, "STREAMS Data Structures" for more info.)
- Always validate `ioc_count` to see whether the `ioctl` is the transparent or `I_STR` form.

Rules for `Put` and `Service` Procedures

To ensure proper data flow between modules and drivers, the following rules should be observed in `put` and `service` procedures:

- `Put` procedures process messages immediately; `service` procedure processing is deferred.
- `Put` and `service` procedures must not sleep.
- Return codes can be sent with STREAMS messages `M_IOCACK`, `M_IOCNAK`, and `M_ERROR`.
- Protect data structures common to `put` and `service` procedures by using (mutex) routines. or perimeter (see the "MT STREAMS perimeters" section of Chapter 13, "Multi-Threaded STREAMS").
- `Put` and `service` procedures cannot access the information in the `uarea` of a process.
- Processing `M_DATA` messages by both `put` and `service` procedures could lead to messages going out of sequence or causing race conditions. The `put` procedure should check if any messages were queued before processing the current message. On the read-side, it is suggested that you have the `put` procedure check if the `service` procedure is running, to avoid the possibility of a race condition. That is, if there are unprotected sections in the `service` procedure, the `put` procedure can be called and run to completion while the `service` procedure is running (the `put` procedure can interrupt the `service` procedure on the read-side). For example, the `service` procedure is running and it removes the last message from the queue, but before it puts the message upstream the `put` procedure is called

(for example, from an interrupt routine) at an unprotected section in the service procedure. The `put` procedure sees that the queue is empty and processes the message. The `put` procedure then returns and the service procedure resumes; but at this point data is out of order because the `put` procedure sent upstream the message that was received after the data the service procedure was processing.

Put Procedures

1. Each queue must define a `put` procedure in its `qinit` structure for passing messages between modules.
2. A `put` procedure must use the `putq()` (see Appendix C, "STREAMS Utilities" for more information) utility to queue a message on its own queue. This is necessary to ensure that the various fields of the `queue` structure are maintained consistently.
3. When passing messages to a neighboring module, a module may not call `putq()` directly, but must call its neighbor module's `put` procedure (see `putnext()` in Appendix C).

However, the `q_qinfo` structure that points to a module's `put` procedure may point to `putq()` (for example, `putq()` is used as the `put` procedure for that module). When a module calls a neighbor module'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 processed by 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.

4. Do not do a `putnext()` to a queue you don't control. Only `putq()` on your own queue or one you do control. The only entry point into another queue is via the STREAMS framework.

Service Procedures

1. If flow control is desired, a service procedure is required. The `canputnext()` or `bcanputnext()` routines should be used by service procedures before doing `putnext()` to honor flow control.
2. The service procedure must use `getq()` to obtain a message from its message queue, so that the flow control mechanism is maintained.

3. The `service` procedure should process all messages on its queue. The only exception is if the Stream ahead is blocked (for example, `canputnext()` fails) or some other failure like buffer allocation failure. 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 other reasons, it must take explicit steps to assure it will be re-enabled.

4. Basic service procedure scheduling involves `qenable()` and `backenable()`. This assures that no messages are lost.
5. The `service` procedure should not put a high priority message back on the queue, because of the possibility of getting into an infinite loop.
6. The `service` procedure must follow the steps below for each message that it processes. STREAMS flow control relies on strict adherence to these steps:
 - a. Remove the next message from the 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 queue. If there is no message, return.
 - b. If all of the following conditions are met:
 - `canputnext()` or `bcanputnext()` fails and
 - the message type is not a high priority type and
 - the message is to be put on the next queue, continue at Step c. Otherwise, continue at Step d.
 - c. The message must be replaced on the head of the queue from which it was removed using `putbq()`. 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.
 - d. If all of the conditions of Step b are not met, the message should not be returned to the queue. It should be processed as necessary. Then, return to Step a.

Data Structures

Only the contents of `q_ptr`, `q_minpsz`, `q_maxpsz`, `q_hiwat`, and `q_lowat` in the `queue` structure may be altered. `q_minpsz`, `q_maxpsz`, `q_hiwat`, and `q_lowat` are set when the module or driver is opened, but they may be modified subsequently via the `strqset()` utility.

Drivers and modules should not change any fields in the `equeue` structure. The only field of the `equeue` structure they are allowed to reference is `eq_bandp`.

Drivers and modules are allowed to change the `qb_hiwat` and `qb_lowat` fields of the `qband` structure via `strqset()`. They may only read the `qb_count`, `qb_first`, `qb_last`, and `qb_flag` fields.

The routines `strqget()` and `strqset()` must be used to get and set the fields associated with the queue. They insulate modules and drivers from changes in the `queue` structure and also enforce the previous rules.

Dynamic Allocation of STREAMS Data Structures

Previous releases of STREAMS statically configured data structures to support a fixed number of Streams, read and write queues, message and data blocks, link block data structures, and Stream event cells. The only way to change this configuration was to reconfigure and reboot the system. Resources were also wasted because data structures were allocated but not necessarily needed.

In SunOS 5.x, STREAMS mechanisms dynamically allocate the following data structures: `stdat`, `queue`, `linkblk`, `strevent`, `dat`, and `msgb`. STREAMS allocates memory to cover these structures as needed.

Dynamic data structure allocation has the advantage of the kernel being initially smaller than a system with static configuration. The performance of the system may also improve because of better memory utilization and added flexibility.

Module Overview

An executing STREAMS module consists of a pair of initialized `queue` structures and a defined set of kernel-level procedures and data structures used to process data, status, and control information. A Stream may have zero or more modules. User processes push (insert) modules on a Stream using the `I_PUSH` `ioctl` and pop (remove) them using the `I_POP` `ioctl`. Pushing and popping of modules happens in a LIFO (Last-In-First-Out) fashion. Modules manipulate messages as they flow through the Stream.

Note that this differs from a module you write as a driver writer. A module you write consists of initialized `qinit` structures, where an executing module consists of initialized `queue` structures.

Module Procedures

STREAMS module procedures (`open`, `close`, `put`, `service`) have already been described in the previous chapters. This section shows some examples and further describes attributes common to module `put` and `service` procedures.

A module's `put` procedure is called by the preceding module, driver, or Stream head, and always before that `queue`'s `service` procedure. The `put` procedure should do any immediate processing (for example, high-priority messages), while the corresponding `service` procedure performs deferred processing.

The `service` procedure is used primarily for performing deferred processing, with a secondary task to implement flow control. Once the `service` procedure is enabled, it may start but not complete before running user-level code. The `put` and `service` procedures must not block because there is no thread synchronization being done.

Code Example 8-1 shows a STREAMS module read-side `put` procedure:

Code Example 8-1 Read- side `put` Procedure

```
static int
modrput(queue_t *q, mblk_t *mp)
{
    struct mod_prv *modptr;

    modptr = (struct mod_prv *) q->q_ptr; /*state info*/

    if (mp->b_datap->db_type >= QPCTL){ /*proc pri msg*/
        putnext(q, mp); /* and pass it on */
        return (0);
    }

    switch(mp->b_datap->db_type) {
    case M_DATA: /* may process message data */
        putq(q, mp); /* queue msg for service procedure */
        return (0);

    case M_PROTO: /* handle protocol control message */
        .
        .
        .

    default:
        putnext(q, mp);
        return (0);
    }
}
```

The preceding code does the following:

- A pointer to a queue defining an instance of the module and a pointer to a message are passed to the `put` procedure.

- The `put` procedure switches on the type of the message. For each message type, the `put` procedure either enqueues the message for further processing by the module `service` procedure, or passes the message to the next module in the Stream.
- High priority messages are typically processed immediately, but not required, by the `put` procedure and passed to the next module.
- Ordinary (or normal) messages are either queued or passed along the Stream.

Code Example 8-2 shows a module write-side `put` procedure:

Code Example 8-2 Write-side `put` Procedure

```
static int
modwput(queue_t *q, mblk_t *mp)
{
    struct mod_prv *modptr;

    modptr = (struct mod_prv *) q->q_ptr; /*state info*/

    if (mp->b_datap->db_type >= QPCTL){ /* proc pri msg */
        putnext(q, mp);                /* and pass it on */
        return (0);
    }

    switch(mp->b_datap->db_type) {
    case M_DATA:                        /* may process message data */
        putq(q, mp); /* queue msg for service procedure or */
        /* pass message along with putnext(q,mp) */
        return (0);

    case M_PROTO:
        .
        .
        .

    case M_IOCTL:                      /* if cmd in msg is recognized */
        /* process message and send back
reply */
        /* else pass message downstream */

    default:
```

Code Example 8-2 Write-side put Procedure

```

        putnext(q, mp);
        return (0);
    }
}

```

The write-side `put` procedure, unlike the read side, may be passed `M_IOCTL` messages. It is up to the module to recognize and process the `ioctl` command, or pass the message downstream if it does not recognize the command.

Code Example 8-3 shows a general scenario employed by the module's service procedure:

Code Example 8-3 Service Procedure

```

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

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

```

The steps are:

- Retrieve the first message from the queue using `getq()`.

- If the message is high priority, process it immediately, and pass it along the Stream.
- Otherwise, the service procedure should use the `canputnext()` utility to determine if the next module or driver that enqueues messages is within acceptable flow-control limits. The `canputnext()` procedure searches the Stream for the next module, driver, or the Stream head with a service procedure. When it reaches one, it looks at the total message space currently being allocated at that queue for enqueued messages. If the amount of space currently used at that queue reaches the high watermark, the `canputnext()` procedure returns false (zero). If the next queue with a service procedure is within acceptable flow-control limits, `canputnext()` returns true (nonzero).
- If `canputnext()` returns false, the service procedure should return the message to its own queue using the `putbq()` procedure. The service procedure can do no further processing at this time, and it should return.
- If `canputnext()` returns true, the service procedure should complete any processing of the message. This may involve retrieving more messages from the queue, allocating and deallocating header and trailer information, and performing control function, for the module.
- When the service procedure is finished processing the message, it calls the `putnext()` procedure to pass the resulting message to the next queue.

These steps are repeated until there are no messages left in the queue (that is, `getq()` returns NULL) or `canputnext()` returns false.

Filter Module Example

The module shown next, *crmod* in Code Example 8-4, 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 of this module are essentially the same as those of the null module presented in Chapter 7, "Overview of Modules and Drivers":

Code Example 8-4 crmod

```
/* Simple filter
 * converts newline -> carriage return, newline
 */
#include <sys/types.h>
```

Code Example 8-4 crmod

```
#include <sys/param.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

static struct module_info minfo =
    { 0x09, "crmod", 0, INFPSZ, 512, 128 };

static int modopen (queue_t*, dev_t*, int, int, cred_t*);
static int modrput (queue_t*, mblk_t*);
static int modwput (queue_t*, mblk_t*);
static int modwsrv (queue_t*);
static int modclose (queue_t*, int, cred_t*);

static struct qinit rinit = {
    modrput, NULL, modopen, modclose, NULL, &minfo, NULL};

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

struct streamtab crmdinfo={ &rinit, &winit, NULL, NULL};
```

The procedure for configuring *crmod* is shown in Appendix E, "Configuration". *stropts.h* includes definitions of flush message options common to user level modules and drivers. *modopen* and *modclose* are unchanged from the null module example shown in Chapter 7, "Overview of Modules and Drivers". *modrput* is like *modput* from the null module.

Note that, in contrast to the null module example, a single *module_info* structure is shared by the read side and write side. The *module_info* includes the flow control high and low watermarks (512 and 128) for the write queue. (Though the same *module_info* is used on the read queue side, the read side has no service procedure so flow control is not used.) The *qinit* contains the service procedure pointer.

The write side put procedure, the beginning of the service procedure, and an example of flushing a queue are shown next:

```
static int
modwput(queue_t *q, mblk_t *mp)
{
```

```

    if (mp->b_datap->db_type >= QPCTL &&
        mp->b_datap->db_type != M_FLUSH)
        putnext(q, mp);
    else
        putq(q, mp); /* Put it on the queue */
    return (0);
}
static int
modwsrv(queue_t *q)
{
    mblk_t *mp;

    while ((mp = getq(q)) != NULL) {
        switch (mp->b_datap->db_type) {
            default:
                if (canputnext(q)) {
                    putnext(q, mp);
                    break;
                } else {
                    putbq(q, mp);
                    return (0);
                }

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

```

modwput, the write put procedure, switches on the message type. High priority messages 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 messages on one or both queues. It can be processed in the put or service procedure.

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

For an `M_FLUSH` message, *modwsrv* checks the first data byte. If `FLUSHW` is set, the write queue is flushed by use of the `flushq()` utility (see Appendix C, “STREAMS Utilities”). `flushq()` takes two arguments, the queue pointer and a flag. The flag indicates what should be flushed, data messages (`FLUSHDATA`)

or everything (FLUSHALL). Data includes M_DATA, M_DELAY, M_PROTO, and M_PCPROTO messages. The choice of what types of messages to flush is module specific.

Ordinary messages will be returned to the queue if canputnext(q) returns false, indicating the downstream path is blocked. The example continues with the remaining part of *modwsrv* processing M_DATA messages:

```
case M_DATA: {
    mblk_t *nbp = NULL;
    mblk_t *next;
    if (!canputnext(q)) {
        putbq(q, mp);
        return (0);
    }
    /* Filter data, appending to queue */
    for (; mp != NULL; mp = next) {
        while (mp->b_rptr < mp->b_wptr) {
            if (*mp->b_rptr == '\n')
                if (!bappend(&nbp, '\r'))
                    goto push;
            if (!bappend(&nbp, *mp->b_rptr))
                goto push;
            mp->b_rptr++;
            continue;
        }
        push:
        if (nbp)
            putnext(q, nbp);
        nbp = NULL;
        if (!canputnext(q)) {
            if (mp->b_rptr >= mp->b_wptr) {
                next = mp->b_cont;
                freeb(mp);
                mp = next;
            }
            if (mp)
                putbq(q, mp);
            return (0);
        }
    } /* while */
    next = mp->b_cont;
    freeb(mp);
    if (nbp)
        putnext(q, nbp);
}
```

```

    }
    }
    return (0);
}

```

The differences in `M_DATA` processing between this and the example in Chapter 5, “Messages” in the section “*Message Allocation and Freeing*” relate to the manner in which the new messages are forwarded and flow controlled. For the purpose of demonstrating alternative means of processing messages, this version creates individual new messages rather than a single message containing multiple message blocks. When a new message block is full, it is immediately forwarded with the `putnext()` procedure rather than being linked into a single, large message (as was done in Chapter 5, “Messages”). 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 (with `canputnext()`) 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.

Flow Control

To support the STREAMS flow control mechanism, modules that use `service` procedures must invoke `canputnext()` before calling `putnext()`, and use appropriate values for the high and low watermarks. If your module has a `service` procedure, it is your responsibility to manage the flow control. If you don’t have a `service` procedure, then there is no need to do anything.

The queue `hiwat` and `lowat` values limit the amount of data that can be placed on a queue. It prevents depletion of buffers in the buffer pool. Flow control is advisory in nature and it can be bypassed. It is managed by high and low watermarks and regulated by utility routines such as `qenable()`. Module flow control is implemented by using the `canputnext()`, `getq()`, `putq()`, `putbq()`, `insq()`, `rmvq()`, and `canputnext()` procedures.

The following scenario takes place normally in flow control:

A driver sends data to a module using the `putnext()` procedure, and the module's `put` procedure queues data using `putq()`. As a result of `putq()`, the service procedure is enabled and will execute at some indeterminate time in the future. When the `service` procedure runs, it retrieves the data by calling the `getq()` utility.

If the module cannot process data at the rate at which the driver is sending the data, the following happens:

When the message is queued, `putq` increments the value of `q_count` by the size of the message and compares the result against the module's high water limit (`q_hiwat`) value for that write queue or read queue. If the count reaches `q_hiwat`, `putq` will set the internal `FULL` indicator for the queue. This will cause messages from upstream in the case of a write side queue or downstream in the case of a read side queue to be halted (`canputnext()` returns `FALSE`) until the queue count drops below `q_lowat`. `getq` decrements the queue count. If the resulting count is below `q_lowat`, `getq` will back-enable and cause the service procedure to be called for any queue which had been blocked.

Note – Flow control does not prevent reaching `q_hiwat` on any given queue. Flow control may exceed its maximum value before `canputnext` detects `QFULL` and flow is stopped.

The next two examples show a line discipline module's flow control. Code Example 8-5 is a read-side line discipline module and the second shows a write side line discipline module. Note that the read side is the same as the write side but without the `M_IOCTL` processing.

Code Example 8-5 Read-side Line Discipline Module

```
/* read side line discipline module flow control */
static mblk_t *read_canon(mblk_t *);

static int
ld_read_srv(
    queue_t *q)                /* pointer to read queue */
{
    mblk_t *mp;                /* original message */
    mblk_t *bp;                /* canonicalized message */

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

Code Example 8-5 Read-side Line Discipline Module

```

switch (mp->b_datap->db_type) { /* type of msg */
case M_DATA: /* data message */
    if (canputnext(q)) {
        bp = read_canon(mp);
        putnext(q, bp);
    } else {
        putbq(q, mp); /* put messagebackinqueue */
        return (0);
    }
    break;

default:
    if (mp->b_datap->db_type >= QPCTL)
        putnext(q, mp); /* high priority message */
    else { /* ordinary message */
        if (canputnext(q))
            putnext(q, mp);
        else {
            putbq(q, mp);
            return (0);
        }
    }
    break;
}

}
return (0);
}

/* write side line discipline module flow control */
static int
ld_write_srv(
    queue_t *q) /* pointer to write queue */
{
    mblk_t *mp; /* original message */
    mblk_t *bp; /* canonicalized message */

    while ((mp = getq(q)) != NULL) {
        switch (mp->b_datap->db_type) { /* type of msg */
        case M_DATA: /* data message */
            if (canputnext(q)) {
                bp = write_canon(mp);
                putnext(q, bp);
            } else {

```

Code Example 8-5 Read-side Line Discipline Module

```

        putbq(q, mp);
        return (0);
    }
    break;

case M_IOCTL:
    ld_ioctl(q, mp);
    break;

default:
    if (mp->b_datap->db_type >= QPCTL)
        putnext(q, mp); /* high priority message */
    else { /* ordinary message */
        if (canputnext(q))
            putnext(q, mp);
        else {
            putbq(q, mp);
            return (0);
        }
    }
    break;
}
}
return (0);
}

```

Design Guidelines

Module developers should follow these guidelines:

- If a module does not understand the message types, the message types must be passed to the next module.
- The module that acts on an `M_IOCTL` message should send an `M_IOCACK` or `M_IOCNAK` message in response to the `ioctl`. If the module does not understand the `ioctl`, it should pass the `M_IOCTL` message to the next module.
- Modules should be designed in such way that they don't pertain to any particular driver but can be used by all drivers.

-
- In general, modules should not require the data in an `M_DATA` message to follow a particular format, such as a specific alignment. This makes it easier to arbitrarily push modules on top of each other in a sensible fashion. Not following this rule may limit module reusability.
 - Filter modules pushed between a service user and a service provider 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.

Also see “Design Guidelines” on page 160 of Chapter 7, “Overview of Modules and Drivers”.

Device Drivers

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

In SunOS 5.x, there are differences between STREAMS drivers and non-STREAMS driver. Though STREAMS drivers can be considered a subset of device drivers in general, only STREAMS-specific information is presented here. For more information on global driver issues and non-STREAMS drivers, see *Writing Device Drivers*.

Overview of Drivers

A device driver is software that provides an interface between the operating system and a device. The driver controls the device in response to requests from the kernel. These requests are issued through the entry points. The driver provides and manages a path for the data to and from the hardware device, and services interrupts issued by the device controller. In STREAMS, drivers are *opened* and modules are *pushed*.

In SunOS 5.x, there are three types of device drivers:

1. Hardware Driver

This type of driver only communicates with a specific piece of hardware. Given the variety of hardware peripherals available, these drivers have many functions.

2. Pseudo Driver

This is configured, installed and acts like a hardware driver, only it does not talk to any hardware.

3. Multiplexer Driver

This is a regular STREAMS driver but has multiple Streams connected to it instead of just one Stream. Multiple connections occur when more than one minor device of the same driver is in use. See Chapter 10, "Multiplexing" for more information.

Unlike a module, a device driver typically has an interrupt routine so that it is accessible from a hardware interrupt as well as from the Stream, unless it is a pseudo driver or a multiplexer driver. However, these particular differences are not recognized by the STREAMS mechanism. They are handled by developer-provided code included in the driver procedures.

The STREAMS framework supports a CLONEOPEN facility. If a STREAMS device driver chooses to support CLONEOPEN, it may be referred to as a cloneable device.

Driver Classification

In general, drivers are grouped according to the *type* of the device they control, the *access* method (the way data is transferred), and the *interface* between the driver and the device.

The type can be hardware or software. A hardware driver controls a physical device, such as a disk. A software driver, also called a pseudo driver, controls software, which in turn may interface with a hardware device. The software driver may also support pseudo devices that have no associated physical device.

Writing a Driver

General Programming

Writing a driver differs from writing other C programs in the following ways:

- A driver does not have a `main` routine. Rather, driver entry points are given specific names and accessed in a variety of ways.
- A driver functions as a part of the kernel. Consequently, a poorly written driver can degrade system performance or corrupt the system.

- A driver cannot use system calls or the C library, because the driver functions at a lower level.
- A driver cannot use floating point arithmetic.
- A driver cannot use archives or shared libraries, but frequently used subroutines can be put in separate files in the source code directory for the driver.

Driver Programming

The following lists rules of driver development:

- Drivers must have `attach(9E)`, `probe(9E)` and `identify(9E)` entry points to initialize the driver. The `attach` routine initializes the driver. Software drivers will usually have little to initialize, because there is no hardware involved.
- Drivers will have `open` and `close` routines.
- Most drivers will have an interrupt handler routine. The driver developer is responsible for supplying an interrupt routine for the device's driver. In addition to hardware interrupts, the system also supports software interrupts. A software interrupt is generated by calling `ddi_trigger_softintr(9F)`.
- All minor nodes are generated by the routine `ddi_create_minor_node(9F)`.

Entry Points

Here are the five entry points through which you can access the driver code:

1. Kernel dynamic loading

These are the routines that allow the kernel to find the driver in the file system and load it into or unload it from the running kernel. These include `_init`, `_fini`, and `_info`.

2. Initialization entry points

These routines are accessed through the `dev_ops` data structure during system initialization. They include `getinfo(9E)`, `identify(9E)`, `probe(9E)`, `attach(9E)`, and `detach(9E)`.

3. Table driven entry points

These routines are accessed through `cb_ops`, the character and block access tables, when the appropriate system call is issued. The `cb_ops` table contains a pointer to the `streamtab` structure.

4. STREAMS queue processing entry points

These routines are pointed to by the `streamtab` and read and process the messages that travel through the queue structures. They include `put`, `srv`, `open`, and `close`.

5. Interrupt routines

These are routines to handle the interrupts for the drivers. They are registered with the `ddi_add_intr(9F)` when the kernel configuration software calls `attach()`. This loads the `ddi_add_intr` routine, which has a pointer to the interrupt handler.

STREAMS Drivers

STREAMS Driver Configuration

As with other SunOS 5.x drivers, STREAMS drivers are dynamically linked, allowing them to be loadable. All drivers are dynamically loaded when referenced for the first time. For example, when the system is initially booted, the `pts` pseudo driver will be loaded automatically into the kernel when it is first accessed.

Note – The word *module* is used in two different ways when talking about drivers. There are STREAMS modules, which are pushable non-driver entities, and there are kernel-loadable modules, which are components of the kernel.

In STREAMS, the header declarations differ between drivers and modules. See Chapter 8, “Modules” and Appendix E, “Configuration”, for more information on how to set up the declarations. Also see the appropriate chapters in the *Writing Device Drivers* manual.

STREAMS Entry Points

STREAMS device drivers have interrupt routines that are callbacks registered with the framework. These entry points are accessed via STREAMS, and the call formats differ from traditional character device drivers. (STREAMS drivers are character drivers, too. The non-STREAMS character drivers are considered traditional character drivers or non-STREAMS character drivers.) The `put` procedure is a driver's entry point, but it is a message (not system) interface. The Stream head translates `write` and `ioctl` calls into messages and sends them downstream to be processed by the driver's write queue `put` procedure. `read` is seen directly only by the Stream head, which contains the functions required to process system calls. A driver does not know about system interfaces other than `open` and `close`, but it can detect the absence of a `read` indirectly if flow control propagates from the Stream head to the driver and affects the driver's ability to send messages upstream.

For read-side processing, when the driver is ready to send data or other information to a user process, it prepares a message and sends it upstream to the read queue of the appropriate (minor device) Stream. The driver's `open` routine generally stores the queue address corresponding to this Stream.

For write-side (or output) processing, the driver receives messages in place of a `write` call. If the message can not be sent immediately to the hardware, it may be stored on the driver's write message queue. Subsequent output interrupts can remove messages from this queue.

Figure 9-1 shows multiple Streams (corresponding to minor devices) connecting to a common driver. There are two distinct Streams opened from the same major device. Consequently, they have the same `streamtab` and the same driver procedures.

The configuration mechanism distinguishes between STREAMS devices and traditional character devices, because system calls to STREAMS drivers are processed by STREAMS routines, not by the system driver routines. In the `cb_ops` structure, the `streamtab` pointer provides this distinction. If it is `NULL` then there are no STREAMS routines to execute. For more detail, see Appendix E, "Configuration".

Multiple instances (minor devices) of the same driver are handled during the initial `open` for each device. Typically, the `queue` address is stored in a driver-private structure "uniquely identified" by the minor device

number. See also `ddi_soft_state` (9F). The `q_ptr` of the `queue` will point to the private data structure entry. When the messages are received by the queue, the calls to the driver `put` and `service` procedures pass the address of the `queue`, allowing the procedures to determine the associated device via the `q_ptr` field.

A driver is at the end of a Stream. As a result, drivers must include standard processing for certain message types that a module might simply be able to pass to the next component.

STREAMS guarantees that only one `open` or `close` can be active at a time per major/minor device pair.

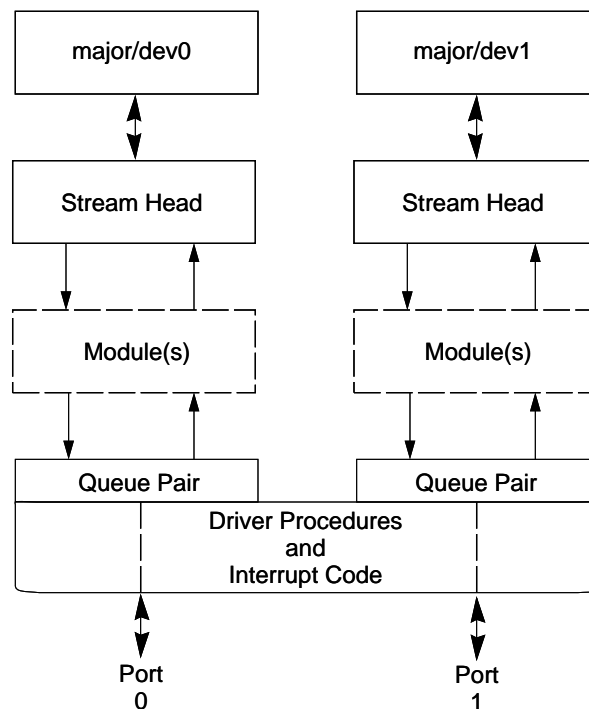


Figure 9-1 Device Driver Streams

Printer Driver Example

The next example 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 device number.

- Flush handling

A driver must loop `M_FLUSH` messages back upstream.

- `ioctl` handling

A driver must send a negative acknowledgment for `ioctl` messages it does not understand. See Chapter 7, “Overview of Modules and Drivers” in the section “Module and Driver `ioctls`” for more discussion.

The driver declarations, Code Example 9-1, follow (see also “Module and Driver Declarations” on page 126):

Code Example 9-1 Driver Declarations

```
/* Simple line printer driver */

#include <sys/types.h>
#include <sys/param.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/signal.h>
#include <sys/errno.h>
#include <sys/cred.h>
#include <sys/stat.h>
#include <sys/modctl.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

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

static int lpopen(queue_t*, dev_t*, int, int, cred_t*);
static int lpclose(queue_t*, int, cred_t*);
static int lpwput(queue_t*, mblk_t*);
```

Code Example 9-1 Driver Declarations

```
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 (('l'<<8)|1)/* 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 *qp;           /* back pointer to write queue */
    kmutex_t lp_lock;      /* sync lock */
};

/* Flags bits */
#define BUSY 1             /* dev is running, int will be forthcoming */

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

static void lpout(struct lp *lp);
```

The `ddi_soft_state(9F)` manual page describes how to maintain multiple instances of a driver.

The values in the module name and ID fields in the `module_info` structure should be unique in the system.

There is no read-side put or service procedure. The flow control limits for use on the write-side are 50 bytes for the low watermark and 150 bytes for the high watermark.

The 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.

lp_lock

A lock to prevent multithread race conditions.

The STREAMS mechanism allows only one Stream per minor device. The driver open routine is called whenever a STREAMS device is opened. It is open's responsibility to assign a private data structure. The driver `open`, `lpopen` in this example, has the same interface as the module `open`:

```
static int lpopen(
    queue_t *q,           /* read queue */
    dev_t *devp,
    int flag,
    int sflag,
    cred_t *credp)
{
    extern lp_cnt;         /* max # of lp devices */
    struct lp *lp;
    minor_t device;

    if (sflag) /* driver refuses to do module or clone open */
        return(ENXIO);

    device = getminor(*devp);
    if (device >= lp_cnt)
        return(ENXIO);

    /* Check if open already. Can't have multiple opens */
    if (q->q_ptr) {
        return(EBUSY);
    }
    lp = &lp_lp[device];
    lp->qptr = WR(q);
    q->q_ptr = (char *) lp;
    WR(q)->q_ptr = (char *) lp;
    qprocson(q);
    return(0);
}
```

The Stream flag, *sflag*, must have the value 0, indicating a normal driver open. *devp* is a pointer to the major/minor device number for this port. After checking *sflag*, the STREAMS open flag, *lopen* extracts the minor device pointed to by *devp*, using the `getminor()` function. *credp* is a pointer to a credentials structure.

The minor device number selects a printer. The device number pointed to by *devp* must be less than *lp_cnt*, the number of configured printers. Otherwise failure occurs.

The next check, `if (q->q_ptr)...`, determines if this printer is already open. If it is, EBUSY is returned to avoid merging printouts 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 before the first open. In this example, the driver sets the value of *q_ptr*, in both the read and write queue structures, to point to a private data structure for the minor device, *lp_lp[device]*.

There are no physical pointers between the read and write queue of a pair. *WR* is a queue pointer function. *WR(q)* generates the write pointer from the read pointer. *RD* and *OTHER* are additional queue pointer functions. *RD(q)* generates the read pointer from the write pointer, and *OTHER(q)* generates the mate pointer from either. With the DDI, *WR*, *RD*, and *OTHER* are now functions, not macros.

Driver Flush Handling

The following write `put` procedure, *lpwput*, illustrates driver `M_FLUSH` handling. Note that all drivers are expected to incorporate flush handling.

If `FLUSHW` is set, the write message queue is flushed, and (in this example) the leading message (*lp->msg*) is also flushed. *lp_lock* protects the drivers per instance data structure. Note: there is only one lock for all instances of this driver for the sake of simplicity.

Normally, if `FLUSHR` is set, the read queue would be flushed. However, in this example, no messages are ever placed on the read queue, so it is not necessary to flush it. The `FLUSHW` bit is cleared and the message is sent upstream using `qreply()`. 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 queues `M_DATA` and `M_IOCTL` messages and, if the device is not busy, starts output by calling *lpout*. Messages types that are not recognized are discarded.

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

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

    switch (mp->b_datap->db_type) {
    default:
        freemsg(mp);
        break;

    case M_FLUSH: /* Canonical flush handling */
        if (*mp->b_rptr & FLUSHW) {
            flushq(q, FLUSHDATA);
            mutex_enter(&lp->lp_lock); /* lock any access to
lp */
            if (lp->msg) {
                freemsg(lp->msg);
                lp->msg = NULL;
            }
            mutex_exit(&lp->lp_lock);
        }
        if (*mp->b_rptr & FLUSHR) {
            *mp->b_rptr &= ~FLUSHW;
            greply(q, mp);
        } else
            freemsg(mp);
        break;

    case M_IOCTL:
    case M_DATA:
```

```

        putq(q, mp);
        mutex_enter(&lp->lp_lock);
        if (!(lp->flags & BUSY))
            lpout(lp);
        mutex_exit(&lp->lp_lock);
    }
    return (0);
}

```

Driver Interrupt

The following example shows the interrupt routine in the printer driver.

`lpint` is the driver interrupt handler routine.

`lpout` takes a character from the queue and sends it to the printer. For convenience, the message currently being output is stored in `lp->msg`. It is assumed that this is called with the mutex held.

`lpoutchar` sends a character to the printer and interrupts when complete. Printer interface options need to be set before being able to print.

```

/* Device interrupt routine */

static int
lpint(
    caddr_t intr_arg)          /* minor device number of lp */
{
    struct lp *lp;
    minor_t device = (minor_t) intr_arg;
    lp = &lp_lp[device];
    mutex_enter(&lp->lp_lock);

    if (!(lp->flags & BUSY)) {
        mutex_exit(&lp->lp_lock);
        return (DDI_INTR_UNCLAIMED);
    }
    lp->flags &= ~BUSY;
    lpout(lp);
    mutex_exit(&lp->lp_lock);
    return (DDI_INTR_CLAIMED);
}

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

```

```

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

    q = lp->qptr;

loop:
    if ((bp = lp->msg) == NULL) { /*no current message*/
        if ((bp = getq(q)) == NULL) {
            lp->flags &= ~BUSY;
            return;
        }
        if (bp->b_datap->db_type == M_IOCTL) {
            lpdoioctl(lp, bp);
            goto loop;
        }
        lp->msg = bp; /* new message */
    }
    if (bp->b_rptr >= bp->b_wptr) { /* validate message */
        bp = lp->msg->b_cont;
        lp->msg->b_cont = NULL;
        freeb(lp->msg);
        lp->msg = bp;
        goto loop;
    }

    lpoutchar(lp, *bp->b_rptr++); /*output one character*/
    lp->flags |= BUSY;
}

```

Driver Close

The driver close routine is called by the Stream head. Any messages left in the queue will be automatically removed by STREAMS. The Stream is dismantled data structures are released.

```
static int
lpclose(
    queue_t *q,           /* read queue */
    int flag,
    cred_t *credp)
{
    struct lp *lp;

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

    /* Free message, queue is automatically
     * flushed by STREAMS */

    mutex_enter(&lp->lp_lock);
    if (lp->msg) {
        freemsg(lp->msg);
        lp->msg = NULL;
    }
    lp->flags = 0;
    mutex_exit(&lp->lp_lock);
}
```

Driver Flow Control

The same utilities (described in Chapter 8, "Modules") and mechanisms used for module flow control are used by drivers.

When the message is queued, `putq()` increments the value of `q_count` by the size of the message and compares the result against the driver's write high watermark (`q_hiwat`) value. If the count reaches `q_hiwat`, the `putq()` utility routine will set the internal FULL indicator for the driver write queue. This will cause messages from upstream to be halted (`canputnext()` returns FALSE) until the write queue count drops below `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`, the `getq()` routine will back-enable any upstream queue that had been blocked.

For priority band data, `qb_count`, `qb_hiwat`, and `qb_lowat` are used.

STREAMS allows flow control to be used on the driver read-side to handle temporary upstream blocks.

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

Cloning

In many earlier examples, each user process connected a Stream to a driver by opening a particular minor device of that driver. Often, however, there is a need for a user process to connect a new Stream to a driver regardless of which minor device is used to access the driver. In the past, this 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 device that the driver controls. 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 major/minor device, so the total number of Streams that may be connected to a particular 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.

Note – The decision to implement a STREAMS driver as a cloneable device is made by the designers of the device driver. Knowledge of clone driver implementation is not required to use it.

There are two ways to open as a clone device. The first is by having a `CLONEOPEN` flag passed in, the result of which is presented in the following example. The second way is to have the driver open itself that way.

For the ptm device, the first technique is:

The module `_init` routine sets up `dev_ops` to point to the `attach` routine, and the `cb_ops` to point to the `open` routine through the `streamtab`. The `attach` routine shown below creates the device file in `/devices`, which has a clone major number of 11 and a minor of 23, the major number of the device driver:

crw-rw-rw-	1 sys	11, 23 Mar	6 02:05	clone:ptmx
crw-----	1 sys	23, 0 Mar	6 02:05	ptm:ptmajor

When the file `/devices/pseudo/clone@0:ptmx` is opened, the clone code

```
static int
ptm_attach(dev_info_t *devi, ddi_attach_cmd_t cmd)
{
    if (cmd != DDI_ATTACH)
        return (DDI_FAILURE);

    if (ddi_create_minor_node(devi, "ptmajor", S_IFCHR,
        0, NULL, 0) == DDI_FAILURE) {
        ddi_remove_minor_node(devi, NULL);
        return (DDI_FAILURE);
    }

    if (ddi_create_minor_node(devi, "ptmx", S_IFCHR,
        0, NULL, CLONE_DEV) == DDI_FAILURE) {
        ddi_remove_minor_node(devi, NULL);
        return (DDI_FAILURE);
    }

    return (DDI_SUCCESS);
}
```

in the kernel (accessed by major 11) passes the `CLONEOPEN` flag to the `ptmopen` routine. `ptm` then checks `sflag` to make sure it is a clone driver:

```
static int
ptmopen(rqp, devp, oflag, sflag, credp)
    queue_t *rqp; /* pointer to the read-side queue */
    dev_t *devp; /* pointer to stream tail's dev */
    int oflag; /* the user open(2) supplied flags */
    int sflag; /* open state flag */
    credp;
```

```

    cred_t *credp; /* credentials */
{
    struct pt_ttys    *ptmp;
    mblk_t            *mop;
    dev_t              dev;

    if (sflag != CLONEOPEN) {
        cmn_err(CE_WARN, "ptmopen: invalid sflag\n");
        return (EINVAL);
    }
}

```

For the second technique, the log driver will show how it opens a clone device itself. The attach routine is much like the one in the preceding example.

```

#define    CONSWMIN 0
#define    CLONEMIN 5

static int
log_attach(dev_info_t *devi, ddi_attach_cmd_t cmd)
{
    if (ddi_create_minor_node(devi, "conslog", S_IFCHR,
        CONSWMIN, NULL, 0) == DDI_FAILURE ||
        ddi_create_minor_node(devi, "log", S_IFCHR,
        CLONEMIN, NULL, 0) == DDI_FAILURE) {
        ddi_remove_minor_node(devi, NULL);
        return (DDI_FAILURE);
    }
    return (DDI_SUCCESS);
}

```

But now when the open routine is run, there is special cloning logic in the driver to handle it:

```

static int
logopen(
    queue_t            *q,
    dev_t              *devp,
    int                 flag,
    int                 sflag,
    cred_t              *cr)
{
    int i;
    struct log *lp;
    /*
     * A MODOPEN is invalid and so is a CLONEOPEN.

```

```

    */
    if (sflag)
        return (ENXIO);

    mutex_enter(&log_lock);
    switch (getminor(*devp)) {

    case CONSWMIN:
        if (flag & FREAD) { /* can only write to this minor */
            mutex_exit(&log_lock);
            return (EINVAL);
        }
        if (q->q_ptr) { /* already open */
            mutex_exit(&log_log);
            return (0);
        }
        lp = &log_log[CONSWMIN];
        break;

    case CLONEMIN:
        /*
         * Find an unused minor > CLONEMIN.
         */
        i = CLONEMIN;
        for (lp = &log_log[i]; i < log_cnt; i++, lp++) {
            if (!(lp->log_state & LOGOPEN))
                break;
        }
        if (i >= log_cnt) {
            mutex_exit(&log_lock);
            return (ENXIO);
        }
        /* clone it */
        *devp=makedevice(getmajor(*devp) (minor_t)i);
        break;
    }
}

```

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 almost like a full-duplex pipe. The Streams are not physically linked. The driver is a simple multiplexer that passes messages from one Stream's write queue to the other Stream's read queue.

To create a connection, a process opens two Streams, obtains the minor device number associated with one of the returned file descriptors, and sends the device number in an `ioctl(2)` to the other Stream. For each `open`, the driver `open` places the passed `queue` pointer in a driver interconnection table, indexed by the device number. When the driver later receives an `M_IOCTL` message, it uses the device number to locate the other Stream's interconnection table entry, and stores the appropriate `queue` pointers in both of the Streams' interconnection table entries.

Subsequently, when messages other than `M_IOCTL` or `M_FLUSH` are received by the driver on either Stream's write-side, the messages are switched to the read queue following the driver on the other Stream's read-side. The resultant logical connection is shown in Figure 9-2. Flow control between the two Streams must be handled explicitly, since STREAMS will not automatically propagate flow control information between two Streams that are not physically connected.

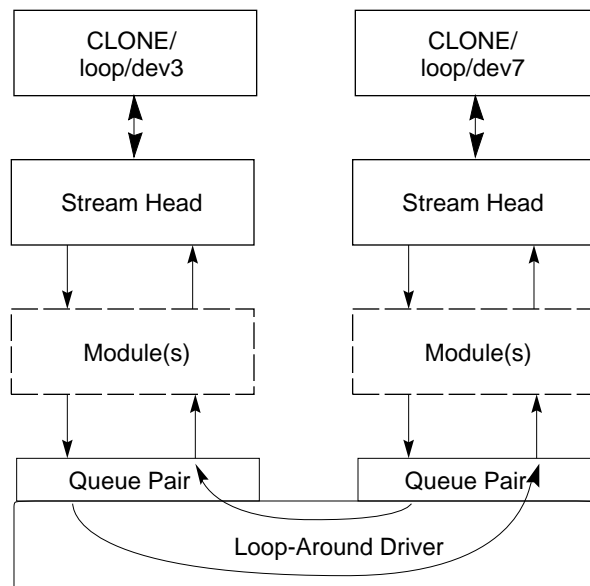


Figure 9-2 Loop-Around Streams

The next example shows the loop-around driver code. The *loop* structure contains the interconnection information for a pair of Streams. *loop_loop* is indexed by the minor device number. When a Stream is opened to the driver, the arriver places the address of the corresponding *loop_loop* element in *q_ptr* (private data structure pointer) of the read-side and write-side queues. Since STREAMS clears *q_ptr* when the queue is allocated, a NULL value of *q_ptr* indicates an initial open. *loop_loop* is used to verify that this Stream is connected to another open Stream.

Note – The code presented here for the loop-around driver represents a single threaded, uni-processor implementation. Multi-processor and multithreading issues such as locking for data corruption and to prevent race conditions are not discussed. See Chapter 13, “Multi-Threaded STREAMS” for details.

The declarations for the driver are:

```
/* Loop-around driver */

#include <sys/types.h>
#include <sys/param.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/signal.h>
#include <sys/errno.h>
#include <sys/cred.h>
#include <sys/stat.h>
#include <sys/modctl.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

static struct module_info minfo = {
    0xeel2, "loop", 0, INFPSZ, 512, 128 };

static int loopopen (queue_t*, dev_t*, int, int, cred_t*);
static int loopclose (queue_t*, int, cred_t*);
static int loopwput (queue_t*, mblk_t*);
static int loopwsrv (queue_t*);
static int looprsrv (queue_t*);
static void loopcopy(mblk *, mblk_t *, uint, unsigned char);

static struct qinit rinit = {
```

```

    NULL, looprsrv, loopopen, loopclose, NULL, &minfo, NULL};

static struct ginit 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 (('l'<8)|1) /* in a .h file */
extern struct loop loop_loop[];
extern int loop_cnt;

```

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

```

static int loopopen(
    queue_t *q,
    dev_t *devp,
    int flag,
    int sflag,
    cred_t *credp)
{
    struct loop *loop;

    minor_t newminor;

    if (q->q_ptr) /* already open */
        return(0);

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

```

```

    if (newminor >= loop_cnt)
        return(ENXIO);

    /*
     * construct new device number and reset devp
     * getmajor gets the major number
     */

    *devp = makedevice(getmajor(*devp), newminor);
    loop = &loop_loop[newminor];
    WR(q)->q_ptr = (char *) loop;
    q->q_ptr = (char *) loop;
    loop->qptr = WR(q);
    loop->oqptr = NULL;

    qprocson(q);
    return(0);
}

```

In *loopopen*, *sflag* can be CLONEOPEN, indicating that the driver should pick an unused minor device (that is, 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 specified by *getminor(*devp)* is used.

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(queue_t *q, mblk_t *mp)
{
    struct loop *loop;
    int to;
    loop = (struct loop *)q->q_ptr;

    switch (mp->b_datap->db_type) {

        case M_IOCTL: {
            struct iocblk          *iocp;
            int                     error=0;
            iocp = (struct iocblk *)mp->b_rptr;

            switch (iocp->ioc_cmd) {

```



```

case LOOP_SET: {

    /*
     * if this is a transparent ioctl then convert the
     * message into an M_COPYIN message so that the
     * data will ultimately be copied from user space
     * to kernel space.
     */
    if (iocp->ioc_count == TRANSPARENT) {
        loopcopy(mp, (mblk_t *)NULL,
            sizeof (struct loop), M_COPYIN);
        qreply(q, mp);
        break; /* leave LOOP_SET case */
    }

    /* fetch other minor device number */
    to = *(int *)mp->b_cont->b_rptr;

    /*
     * Sanity check.  ioc_count contains the amount
     * of user supplied data which must equal the
     * size of an int.
     */
    if (iocp->ioc_count != sizeof(int)) {
        error = EINVAL;
        goto iocnak;
    }

    /* Is the minor device number in range? */
    if (to >= loop_cnt || to < 0) {
        error = ENXIO;
        goto iocnak;
    }

    /* Is the other device open? */
    if (!loop_loop[to].qptra) {
        error = ENXIO;
        goto iocnak;
    }

    /* Check if either dev is currently connected */
    if (loop->oqptra || loop_loop[to].oqptra) {
        error = EBUSY;
        goto iocnak;
    }
}

```

```

    }

    /* Cross connect the streams via the loopstruct */
    loop->oqp_ptr = RD(loop_loop[to].qp_ptr);
    loop_loop[to].oqp_ptr = RD(q);

    /*
     * Return successful ioctl. Set ioc_count
     * to zero, since no data is returned.
     */
    mp->b_datap->db_type = M_IOCACK;
    iocp->ioc_count = 0;
    qreply(q, mp);
    break;
}

default:
    error = EINVAL;

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

/*
 * Convert mp to an M_COPYIN or M_COPYOUT message (as specified
 * by type) requesting size bytes. Assumes mp denotes a
 * TRANSPARENT M_IOCTL or M_IOCDATA message. If dp is
 * non-NULL, it is assumed to point to data to be
 * copied out and is linked onto mp.
 */
static void
loopcopy(mblk_t *mp, mblk_t *dp, uint size, unsigned char type)
{

```

```

struct copyreq *cp = (struct copyreq *)mp->b_rptr;

cp->cq_private = NULL;
cp->cq_flag = 0;
cp->cq_size = size;

cp->cq_addr = (caddr_t)((long *) (mp->b_cont->b_rptr));
if (mp->b_cont != NULL)
    freeb(mp->b_cont);

if (dp != NULL) {
    mp->b_cont = dp;
    dp->b_wptr += size;
} else
    mp->b_cont = NULL;

mp->b_datap->db_type = type;
mp->b_wptr = mp->b_rptr + sizeof (*cp);
}

```

loopwput shows another use of an `ioctl` call (see Chapter 7, “Overview of Modules and Drivers” in the section “*Module and Driver ioctls*”). 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 `M_IOCTL` message holds an integer that specifies the minor device number of the Stream to which to connect.

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 it passes these sanity checks, the read queue 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, FLUSHALL);           /* write */
        flushq(loop->oqptr, FLUSHALL);
        /* read on other side equals write on this side */
    }
    if (*mp->b_rptr & FLUSHR) {
        flushq(RD(q), FLUSHALL);
        flushq(WR(loop->oqptr), FLUSHALL);
    }
    switch(*mp->b_rptr) {
    case FLUSHW:
        *mp->b_rptr = FLUSHR;
        break;
    case FLUSHR:
        *mp->b_rptr = FLUSHW;
        break;
    }
    putnext(loop->oqptr, mp);
    break;
default: /* If this Stream isn't connected,
         * send M_ERROR upstream. */
    if (loop->oqptr == NULL) {
        freemsg(mp);
        (void) putnextctl1(RD(q), M_ERROR, ENXIO);
        break;
    }
    putq(q, mp);
}
return (0);
}

```

Finally, *loopwput* queues all other messages (for example, `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.

Certain message types can be sent upstream by drivers and modules to the Stream head where they are translated into actions detectable by user processes. The messages may also modify the state of the Stream head:

M_ERROR

Causes the Stream head to lock up. Message transmission between Stream and user processes is terminated. All subsequent system calls except `close(2)` and `poll(2)` 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. This can also result in the `SIGHUP` signal being sent to the process group's session leader.

M_SIG/M_PCSIG

Causes a specified signal to be sent to the process group associated with the stream.

`putnextctl0()` and `putnextctl1()` are utilities that allocate a non-data (that is, not `M_DATA`, `M_DELAY`, `M_PROTO`, or `M_PCPROTO`) type message, place one byte in the message (for `putctl1next`) and call the `put` procedure of the specified queue.

Service procedures are required in this example on both the write-side and read-side for flow control:

```
static int loopwsrv(queue_t *q)
{
    mblk_t *mp;
    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 &
            !canputnext(loop->oqptr)) {
            putbq(q, mp); /* read-side is blocked */
            break;
        }
        /*
         * send message to queue following
         * other Stream read queue
         */
        putnext(loop->oqptr, mp);
    }
}
```

```

    }
    return (0);
}

static int looprsrv(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 (0);
    /* manually enable write service procedure */
    qenable(WR(loop->oqptr));
    return (0);
}

```

The write service procedure, *loopwsrv*, takes on the canonical form. 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. *loopwsrv* can not be directly back-enabled by STREAMS because there is no direct queue linkage between the two Streams. Note that no message is queued to the read service procedure. Messages are kept on the write-side so that flow control can propagate up to the Stream head. The *qenable()* routine schedules the write-side service procedure of the other Stream.

loopclose breaks the connection between the Streams:

```

static int loopclose(
    queue_t *q,
    int flag,
    cred_t *credp)
{
    struct loop *loop;

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

```

```

/* If we are connected to another stream, break the
 * linkage, and send a hangup message.
 * The hangup message causes the stream head to reject
 * writes, allow the queued data to be read completely,
 * and then return EOF on subsequent reads.
 */
if (loop->oqptr) {
    (void) putnextctl(loop->oqptr, M_HANGUP);
    loop->oqptr = NULL;
}
qprocsoff(q);
((struct loop *)loop->oqptr->q_ptr)->oqptr = NULL;
return (0);
}

```

loopclose sends an `M_HANGUP` message up the connected Stream to the Stream head.

Design Guidelines

Driver developers should follow these guidelines:

- Messages that are not understood by the drivers should be freed.
- A driver must process all `M_IOCTL` messages. Otherwise, the Stream head will block for an `M_IOCNAK`, `M_IOCACK`, or until the timeout (potentially infinite) expires.
- If a driver does not understand an `ioctl`, an `M_IOCNAK` message must be sent to upstream.
- The Stream head locks up the Stream when it receives an `M_ERROR` message, so driver developers should be careful when using the `M_ERROR` message.
- A hardware driver must have an interrupt routine.
- Multithreaded drivers need to protect their own data structures

Also see the section “Design Guidelines” in *Chapter 7, “Overview of Modules and Drivers”*.

Overview of Multiplexing

This chapter describes how STREAMS multiplexing configurations are created and also discusses multiplexing drivers. A STREAMS multiplexer is a driver with multiple Streams connected to it. The primary function of the multiplexing driver is to switch messages among the connected Streams. Multiplexer configurations are created from user level by system calls.

STREAMS-related system calls are used to set up the “plumbing,” or Stream interconnections, for multiplexing drivers. The subset of these calls that allows a user to connect (and disconnect) Streams below a driver is referred to as the multiplexing facility. This type of connection is referred to as a one-to-M, or lower, multiplexer configuration. This configuration must always contain a multiplexing driver, which is recognized by STREAMS as having special characteristics.

Multiple Streams can be connected above a driver by use of `open(2)` calls. This was done for the loop-around driver and for the driver handling multiple minor devices in Chapter 9, “Drivers”. There is no difference between the connections to these drivers, only the functions performed by the driver are different. In the multiplexing case, the driver routes data between multiple Streams. In the device driver case, the driver routes data between user processes and associated physical ports. Multiplexing with Streams connected above is referred to as an N-to-1, or upper, multiplexer. STREAMS does not provide any facilities beyond `open(2)` and `close(2)` to connect or disconnect upper Streams for multiplexing.

From the driver's perspective, upper and lower configurations differ only in the way they are initially connected to the driver. The implementation requirements are the same: route the data and handle flow control. All multiplexer drivers require special developer-provided software to perform the multiplexing data routing and to handle flow control. STREAMS does not directly support flow control among multiplexed Streams.

M-to-N multiplexing configurations are implemented by using both of the above mechanisms in a driver.

As discussed in Chapter 9, "Drivers", the multiple Streams that represent minor devices are actually distinct Streams in which the driver keeps track of each Stream attached to it. The STREAMS subsystem does not recognize any relationship between the Streams. The same is true for STREAMS multiplexers of any configuration. The multiplexed Streams are distinct and the driver must be implemented to do most of the work.

In addition to upper and lower multiplexers, more complex configurations can be created by connecting Streams containing multiplexers to other multiplexer drivers. With such a diversity of needs for multiplexers, it is not possible to provide general-purpose multiplexer drivers. Rather, STREAMS provides a general purpose multiplexing facility. The facility allows users to set up the inter-module/driver plumbing to create multiplexer configurations of generally unlimited interconnection.

Building a Multiplexer

This section builds a protocol multiplexer with the multiplexing configuration shown in Figure 10-1. To free users from the need to know about the underlying protocol structure, a user-level daemon process will be built to maintain the multiplexing configuration. Users can then access the transport protocol directly by opening the transport protocol (TP) driver device node.

An internetworking protocol driver (IP) routes data from a single upper Stream to one of two lower Streams. This driver supports two STREAMS connections beneath it. These connections are to two distinct networks; one for the IEEE 802.3 standard via the 802.3 driver, and other to the IEEE 802.4 standard via the 802.4 driver. The TP driver multiplexes upper Streams over a single Stream to the IP driver.

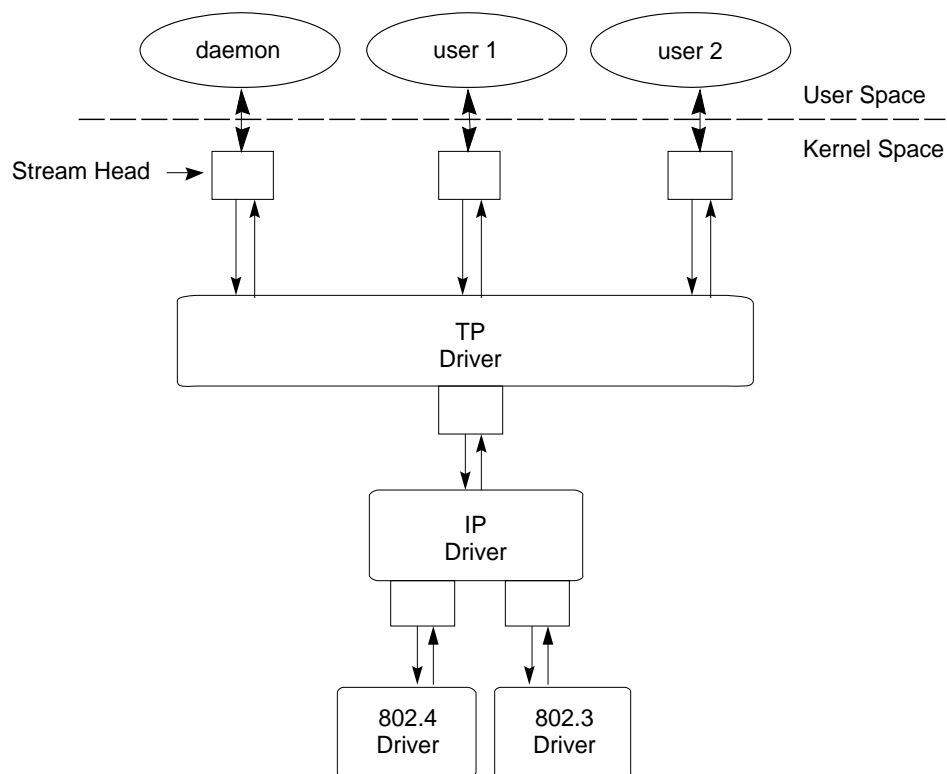


Figure 10-1 MultiplexerProtocol Multiplexer

Code Example 10-1 shows how this daemon process sets up the protocol multiplexer. The necessary declarations and initialization for the daemon program follow:

Code Example 10-1 Protocol Daemon

```
#include <fcntl.h>
#include <stropts.h>
void
main()
{
    int          fd_802_4,
```

Code Example 10-1 Protocol Daemon

```

        fd_802_3,
        fd_ip,
        fd_tp;
/* daemon-ize this process */

switch (fork()) {
case 0:
    break;
case -1:
    perror("fork failed");
    exit(2);
default:
    exit(0);
}
(void)setsid();

```

This multi-level multiplexed Stream configuration will be built from the bottom up. Therefore, the example begins by first constructing the Internal Protocol (IP) multiplexer. This multiplexing device driver is treated like any other software driver. It owns a node in the Solaris file system and is opened just like any other STREAMS device driver.

The first step is to open the multiplexing driver and the 802.4 driver, thus creating separate Streams above each driver as shown in Figure 10-2. The Stream to the 802.4 driver may now be connected below the multiplexing IP driver using the `I_LINK` ioctl call.

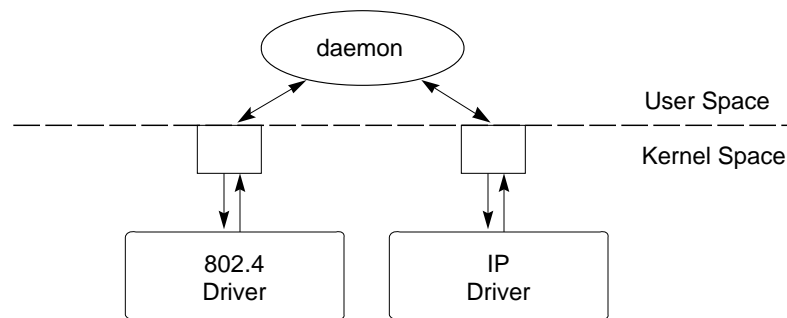


Figure 10-2 Before Link

The sequence of instructions to this point is:

```
if ((fd_802_4 = open("/dev/802_4", O_RDWR)) < 0) {
    perror("open of /dev/802_4 failed");
    exit(1);
}
if ((fd_ip = open("/dev/ip", O_RDWR)) < 0) {
    perror("open of /dev/ip failed");
    exit(2);
}
/* now link 802.4 to underside of IP */
if (ioctl(fd_ip, I_LINK, fd_802_4) < 0) {
    perror("I_LINK ioctl failed");
    exit(3);
}
```

`I_LINK` takes two file descriptors as arguments. The first file descriptor, *fd_ip*, 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 10-3 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. The Stream head's queues of the 802.4 driver will be used by the IP driver to manage the lower half of the multiplexer.

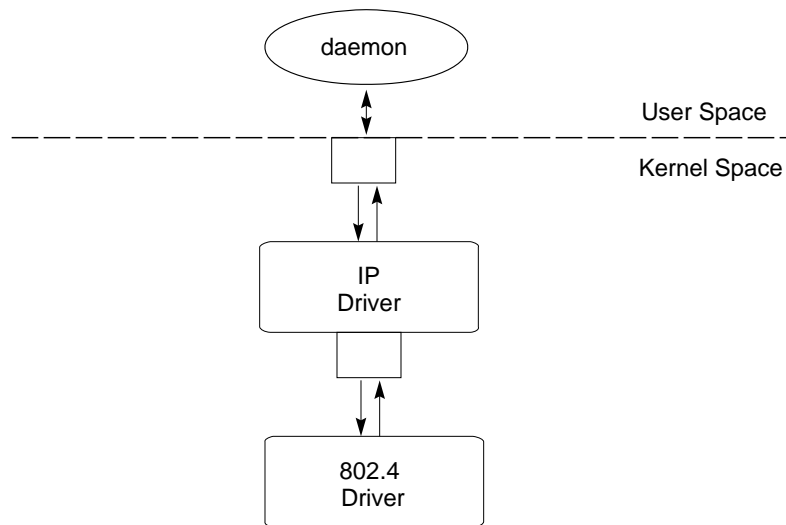


Figure 10-3 IP Multiplexer After First Link

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

The following sequence of system calls is used to continue building the internetworking protocol multiplexer (IP):

```

if ((fd_802_3 = open("/dev/802_3", O_RDWR)) < 0) {
    perror("open of /dev/802_3 failed");
    exit(4);
}
if (ioctl(fd_ip, I_LINK, fd_802_3) < 0) {
    perror("I_LINK ioctl failed");
    exit(5);
}

```

All links below the IP driver have now been established, giving the configuration in Figure 10-4

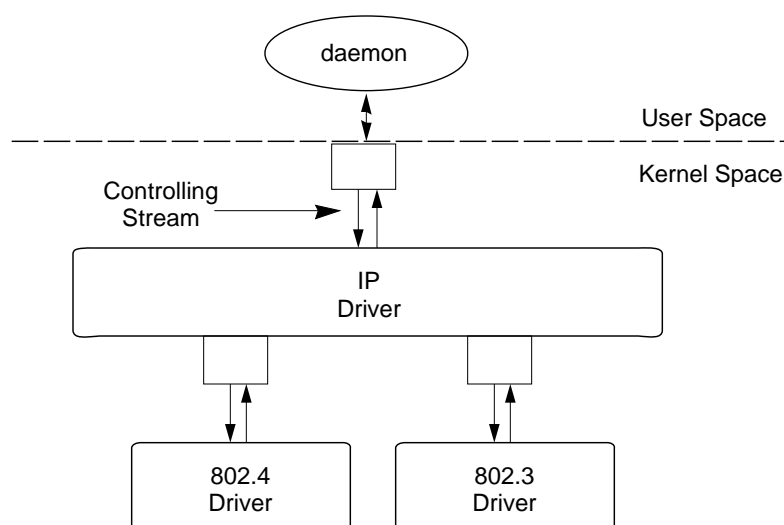


Figure 10-4 IP Multiplexer

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. This 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 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) describes such restrictions. However, only one `I_LINK` operation is allowed for each lower Stream; a single Stream cannot be linked below two multiplexers simultaneously.

Continuing with the example, the IP driver will now be linked below the transport protocol (TP) multiplexing driver. As seen in Figure 10-1, 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 multi-level multiplexing configuration shown in Figure 10-5 has now been created.

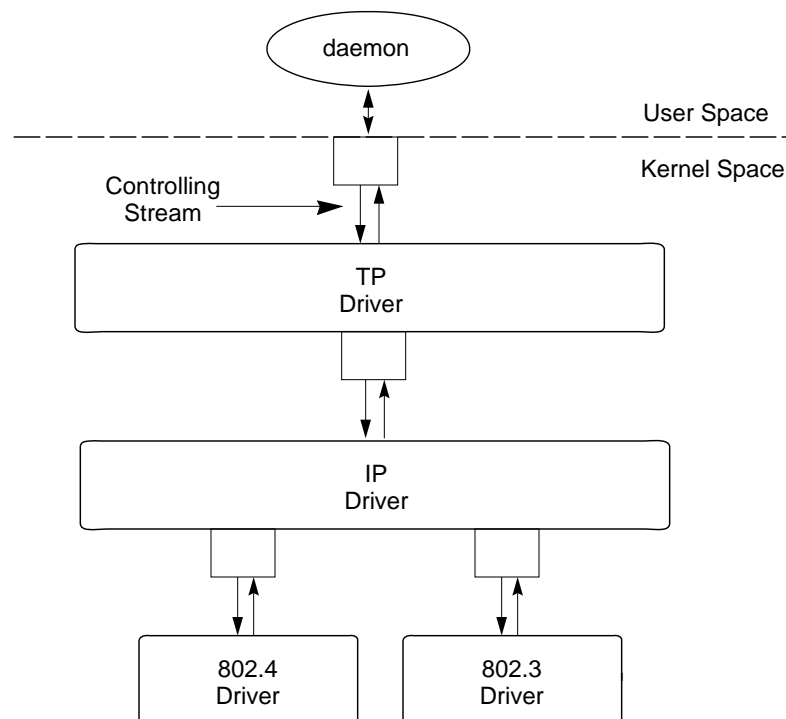


Figure 10-5 TP Multiplexer

Because the controlling Stream of the IP multiplexer has been linked below the TP multiplexer, the controlling Stream for the new multi-level multiplexer configuration is the Stream above the TP multiplexer.

At this point the file descriptors associated with the lower drivers can be closed without affecting the operation of the multiplexer. If these file descriptors are not closed, all subsequent `read`, `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 completes the daemon example:

```
close(fd_802_4);
close(fd_802_3);
close(fd_ip);
/* Hold multiplexer open forever or at least til this process
   is terminated by an external UNIX signal */
pause();
}
```

To summarize, Figure 10-5 shows the multi-level protocol multiplexer. The transport driver supports several simultaneous Streams. These Streams are multiplexed over the single Stream connected to the IP multiplexer. The mechanism for establishing multiple Streams above the transport multiplexer is actually a by-product of the way in which Streams are created between a user process and a driver. By opening different minor devices of a STREAMS driver, separate Streams will be connected to that driver. The driver must be designed with the intelligence to route data from the single lower Stream to the appropriate upper Stream.

The daemon process maintains the multiplexed Stream configuration through an open Stream (the controlling Stream) to the transport driver. Meanwhile, other users can access the services of the transport protocol by opening new Streams to the transport driver; they are freed from the need for any unnecessary knowledge of the underlying protocol configurations and sub-networks that support the transport service.

Multi-level multiplexing configurations should be assembled from the bottom up. That is because the passing of `ioctls` through the multiplexer is determined by the nature of the multiplexing driver and cannot generally be relied on.

Dismantling a Multiplexer

Streams connected to a multiplexing driver from above with `open`, can be dismantled by closing each Stream with `close`. The mechanism for dismantling Streams that have been linked below a multiplexing driver is less obvious, and is described in the following section.

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

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

where *fd* is a file descriptor associated with a Stream connected to the multiplexing driver from above, and *muxid* is the identifier that was returned by `I_LINK` when a driver was linked below the multiplexer. Each lower driver may be disconnected individually in this way, or a special *muxid* value of `MUXID_ALL` may be used to disconnect all drivers from the multiplexer simultaneously.

In the multiplexing daemon program, the multiplexer is never explicitly dismantled. That is because all links associated with a multiplexing driver are automatically dismantled when the controlling Stream associated with that multiplexer is closed. Because the controlling Stream is open to a driver, only the final call of `close` for that Stream will close it. In this case, the daemon is the only process that has opened the controlling Stream, so the multiplexing configuration will be dismantled when the daemon exits.

For the automatic dismantling mechanism to work in the multi-level, 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, multi-level configuration. Because the multiplexing program relied on closing the controlling Stream to dismantle the multiplexed Stream configuration instead of using explicit `I_UNLINK` calls, the *muxid* values returned by `I_LINK` could be ignored.

An important side effect of automatic dismantling on the close is that it is not possible for a process to build a multiplexing configuration with `I_LINK` and then exit. That is because `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.

However, if the process uses persistent links via the `I_PLINK` `ioctl` call, the multiplexer configuration would remain intact after the process exits. "Persistent Links" are described later in this chapter.

Routing Data Through a Multiplexer

As demonstrated, STREAMS provides a mechanism for building multiplexed Stream configurations. However, the criteria by which a multiplexer routes data is driver dependent. For example, the protocol multiplexer might use address information found in a protocol header to determine over which sub-network data 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 *muxid* value to determine to which Stream data should be routed (remember that each multiplexer link is associated with a *muxid*). `I_LINK` passes the *muxid* value to the driver and returns this value to the user. The driver can therefore specify that the *muxid* value 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 *muxid* of the desired lower Stream into the first four bytes of each message passed to it. The driver could then match the *muxid* in each message with the *muxid* of each lower Stream, and route the data accordingly.

Connecting / Disconnecting Lower Streams

Multiple Streams are created above a driver/multiplexer by use of the `open` system call on either different minor devices, or on a cloneable device file. Note that any driver that handles more than one minor device is considered an upper multiplexer.

To connect Streams below a multiplexer requires additional software within the multiplexer. The main difference between STREAMS lower multiplexers and STREAMS device drivers are that multiplexers are pseudo-devices and that multiplexers have two additional `qinit` structures, pointed to by fields in the `streamtab` structure: the *lower half read side* `qinit` and the *lower half write side* `qinit`.

The multiplexer is conceptually divided into two parts: the lower half (bottom) and the upper half (top). The multiplexer `queue` structures that have been allocated when the multiplexer was opened, use the usual `qinit` entries from the multiplexer's `streamtab`. This is the same as any open of the STREAMS device. When a lower Stream is linked beneath the multiplexer, the `qinit` structures at the Stream head are substituted by the bottom half `qinit` structures of the multiplexers. Once the linkage is made, the multiplexer

switches messages between upper and lower Streams. When messages reach the top of the lower Stream, they are handled by `put` and `service` routines specified in the bottom half of the multiplexer.

Connecting Lower Streams

A lower multiplexer is connected as follows: the initial `open` to a multiplexing driver creates a Stream, as in any other driver. `open` uses the first two `streamtab` structure entries to create the driver queues. At this point, the only distinguishing characteristic of this Stream are non-NULL entries in the `streamtab` `st_muxrinit` and `st_muxwinit` fields.

These fields are ignored by `open` (see the rightmost Stream in Figure 10-6). 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 10-6). 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)`]. The Stream head points to the Stream head routines as its procedures (known via its `queue`). An `I_LINK` to the upper Stream, referencing the lower Stream, causes STREAMS to modify the contents of the Stream head's queues in the lower Stream. The pointers to the Stream head routines, and other values, in the Stream head's queues are replaced with those contained in the mux fields of the multiplexing driver's `streamtab`. Changing the Stream head routines on the lower Stream means that all subsequent messages sent upstream by the lower Stream's driver will, ultimately, be passed to the `put` procedure designated in `st_muxrinit`, the multiplexing driver. The `I_LINK` also establishes this upper Stream as the control Stream for this lower Stream. STREAMS remembers the relationship between these two Streams until the upper Stream is closed, or the lower Stream is unlinked.

Finally, the Stream head sends an `M_IOCTL` message with `ioc_cmd` set to `I_LINK` to the multiplexing driver. The `M_DATA` part of the `M_IOCTL` contains a `linkblk` structure. The multiplexing driver stores information from the

`linkblk` structure in private storage and returns an `M_IOCACK` message (acknowledgment). `l_index` is returned to the process requesting the `I_LINK`. This value can be used later by the process to disconnect this Stream.

An `I_LINK` is required for each lower Stream connected to the driver. Additional upper Streams can be connected to the multiplexing driver by `open` calls. Any message type can be sent from a lower Stream to user processes along any of the upper Streams. The upper Streams provide the only interface between the user processes and the multiplexer.

Note that no direct data structure linkage is established for the linked Streams. The read queue's `q_next` will be `NULL` and the write queue's `q_next` will point to the first entity on the lower Stream. Messages flowing upstream from a lower driver (a device driver or another multiplexer) will enter the multiplexing driver `put` procedure with `l_qbot` as the `queue` value. The multiplexing driver has to route the messages to the appropriate upper (or lower) Stream. Similarly, a message coming downstream from user space on any upper Stream has to be processed and routed, if required, by the driver.

Also note that the lower Stream (see the headers and file descriptors in Figure 10-7) 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.

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 implementation dependent.

Disconnecting Lower Streams

Dismantling a lower multiplexer is accomplished by disconnecting (unlinking) the lower Streams. Unlinking can be initiated in three ways:

- An `I_UNLINK` `ioctl` referencing a specific Stream
- An `I_UNLINK` indicating all lower Streams

- The last `close` of the control Stream

As in the link, an unlink sends a `linkblk` structure to the driver in an `M_IOCTL` message. The `I_UNLINK` call, which unlinks a single Stream, uses the `L_index` value returned in the `I_LINK` to specify the lower Stream to be unlinked. The latter two calls must designate a file corresponding to a control Stream, which causes all the lower Streams that were previously linked by this control Stream to be unlinked. However, the driver sees a series of individual unlinks.

If no open references exist for a lower Stream, a subsequent unlink will automatically close the Stream. Otherwise, the lower Stream must be closed by `close` 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. Figure 10-6 shows the Streams before their connection to create the multiplexing configuration of Figure 10-7. Multiple upper and lower Streams interface to the multiplexer driver. The user processes of Figure 10-7 are not shown in Figure 10-6.

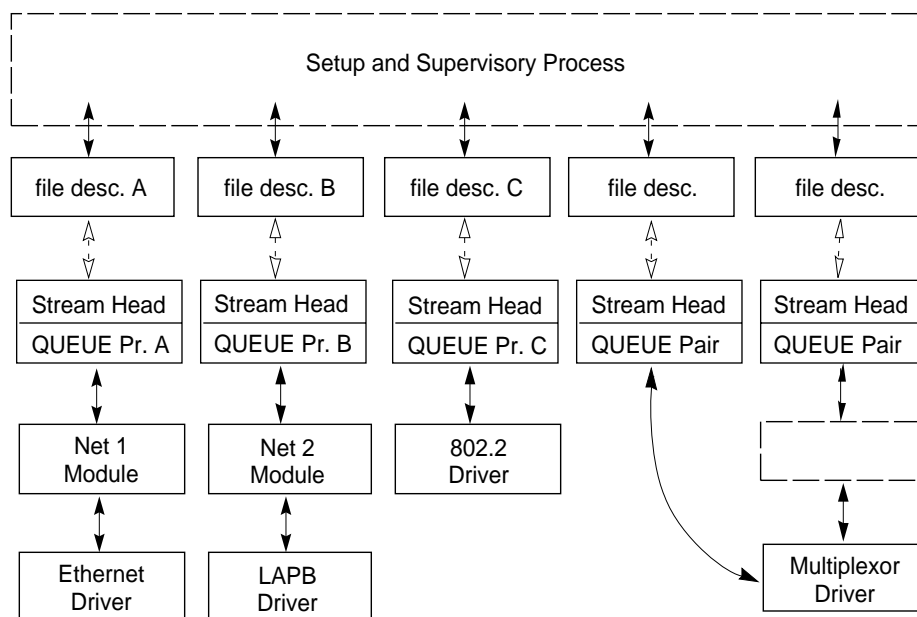


Figure 10-6 Internet Multiplexer Before Connecting

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

Figure 10-6 shows only a portion of the full, larger Stream. In the dotted rectangle above the multiplexer driver, there generally would be an upper transmission control protocol (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.

Figure 10-7 shows that 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 have no further function. (For example, they could be

reconnected through a `I_UNLINK` and `I_LINK` sequence) These lower Streams can be closed to free the file descriptor without any effect on the multiplexer.

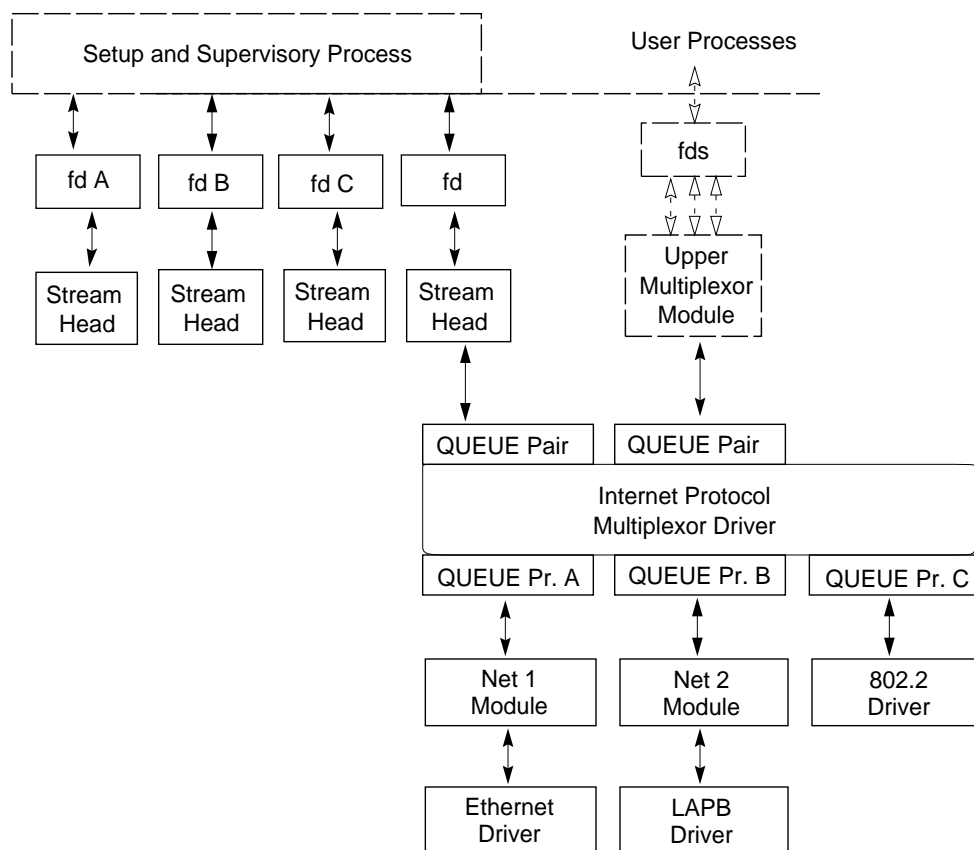


Figure 10-7 Internet Multiplexer After Connecting

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.

Multiplexing Driver

This section contains an example of a multiplexing driver that implements an N-to-1 configuration. This configuration might be used for terminal windows, where each transmission to or from the terminal identifies the window. This resembles a typical device driver, with two differences: the device handling functions are performed by a separate driver, connected as a lower Stream, and the device information (that is, relevant user process) is contained in the input data rather than in an interrupt call.

Each upper Stream is created by `open(2)`. A single lower Stream is opened and then it is linked by use of the multiplexing facility. This lower Stream might connect to the `tty` driver. The implementation of this example is a foundation for an M-to-N multiplexer.

As in the loop-around driver (Chapter 9, "Drivers"), flow control requires the use of standard and special code, since physical connectivity among the Streams is broken at the driver. Different approaches are used for flow control on the lower Stream, for messages coming upstream from the device driver, and on the upper Streams, for messages coming downstream from the user processes.

Note – The code presented here for the multiplexing driver represents a single threaded, uni-processor implementation. Multi-processor and multithreading issues such as locking for data corruption and to prevent race conditions are not discussed. See Chapter 13, "Multi-Threaded STREAMS" for details.

Code Example 10-2 is of multiplexer declarations:

Code Example 10-2 Multiplexer Declarations

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/errno.h>
#include <sys/cred.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

static int muxopen (queue_t*, dev_t*, int, int, cred_t*);
static int muxclose (queue_t*, int, cred_t*);
static int muxuwput (queue_t*, mblk_t*);
static int muxlwsrv (queue_t*);
static int muxlrput (queue_t*, mblk_t*);
static int muxuwsrv (queue_t*);

static struct module_info info = {
    0xaabb, "mux", 0, INFPSZ, 512, 128 };

static struct qinit urinit = { /* upper read */
    NULL, NULL, muxopen, muxclose, NULL, &info, NULL };

static struct qinit uwinit = { /* upper write */
    muxuwput, muxuwsrv, NULL, NULL, NULL, &info, NULL };

static struct qinit lrinit = { /* lower read */
    muxlrput, NULL, NULL, NULL, NULL, &info, NULL };

static struct qinit lwinit = { /* lower write */
    NULL, muxlwsrv, NULL, NULL, NULL, &info, NULL };

struct streamtab muxinfo = {
    &urinit, &uwinit, &lrinit, &lwinit };

struct mux {
    queue_t *qptra; /* back pointer to read queue */
    int     bufcid; /* bufcall return value */
};

extern struct mux mux_mux[];
extern int mux_cnt; /* max number of muxes */
```

Code Example 10-2 Multiplexer Declarations

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

The four `streamtab` entries correspond to the upper read, upper write, lower read, and lower write `qinit` structures. The multiplexing `qinit` structures replace those in each (in this case there is only one) lower Stream head after the `I_LINK` has concluded successfully. In a multiplexing configuration, the processing performed by the multiplexing driver can be partitioned between the upper and lower queues. There must be an upper Stream write `put` procedure and lower Stream read `put` procedure. If the queue procedures of the opposite upper/lower queue are not needed, the queue can be skipped, and the message put to the following queue.

In the example, the upper read side procedures are not used. The lower Stream read queue `put` procedure transfers the message directly to the read queue upstream from the multiplexer. There is no lower write `put` procedure because the upper write `put` procedure directly feeds the lower write queue downstream from the multiplexer.

The driver uses a private data structure, `mux`. `mux_mux[dev]` points back to the opened upper read queue. This is used to route messages coming upstream from the driver to the appropriate upper queue. It is also used to find a free major/minor device for a `CLONEOPEN` driver open case.

Code Example 10-3, the upper queue open, contains the canonical driver open code:

Code Example 10-3 Upper Queue Open

```
static int
muxopen(queue_t *q, dev_t *devp, int flag,
        int sflag, cred_t *credp)
{
    struct mux *mux;
    minor_t device;

    if (q->q_ptr)
```

Code Example 10-3 Upper Queue Open

```

        return(EBUSY);

    if (sflag == CLONEOPEN) {
        for (device = 0; device < mux_cnt; device++)
            if (mux_mux[device].qp_ptr == 0)
                break;

        *devp=makedevice(getmajor(*devp), device);
    }
    else {
        device = getminor(*devp);
        if (device >= mux_cnt)
            return ENXIO;
    }

    mux = &mux_mux[device];
    mux->qp_ptr = q;
    q->q_ptr = (char *) mux;
    WR(q)->q_ptr = (char *) mux;
    qprocson(q);
    return (0);
}

```

muxopen checks for a clone or ordinary open call. It initializes *q_ptr* to point at the *mux_mux[]* structure.

The core multiplexer processing is the following: downstream data written to an upper Stream is queued on the corresponding upper write message queue if the lower Stream is flow controlled. This allows flow control to propagate towards the Stream head for each upper Stream. A lower write *service* procedure, rather than a write *put* procedure, is used so that flow control, coming up from the driver below, may be handled.

On the lower read side, data coming up the lower Stream are passed to the lower read *put* procedure. The procedure routes the data to an upper Stream based on the first byte of the message. This byte holds the minor device number of an upper Stream. The *put* procedure handles flow control by testing the upper Stream at the first upper read queue beyond the driver. That is, the *put* procedure treats the Stream component above the driver as the next queue.

Upper Write Put Procedure

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

```
static int
/*
 * This is our callback routine used by bufcall() to inform us
 * when buffers become available
 */
static void mux_qenable(long ql)
{
    queue_t *q = (queue_t *ql);
    struct mux *mux;

    mux = (struct mux *) (q->q_ptr);
    mux->bufcid = 0;
    qenable(q);
}
muxuwput(queue_t *q, mblk_t *mp)
{
    struct mux *mux;

    mux = (struct mux *) q->q_ptr;
    switch (mp->b_datap->db_type) {
    case M_IOCTL: {
        struct iocblk *iocp;
        struct linkblk *linkp;
        /*
         * ioctl. Only channel 0 can do ioctls. Two
         * calls are recognized: LINK, and UNLINK
         */
        if (mux != mux_mux)
            goto iocnak;

        iocp = (struct iocblk *) mp->b_rptr;
        switch (iocp->ioc_cmd) {
        case I_LINK:
            /*
             * Link. The data contains a linkblk structure
             * Remember the bottom queue in muxbot.
             */
            if (muxbot != NULL)
```

```

        goto iocnak;

        linkp=(struct linkblk *) mp->b_cont->b_rptr;
        muxbot = linkp->l_qbot;
        muxerr = 0;

        mp->b_datap->db_type = M_IOCACK;
        iocp->ioc_count = 0;
        greply(q, mp);
        break;
case I_UNLINK:
    /*
     * Unlink. The data contains a linkblk struct.
     * Should not fail an unlink. Null out muxbot.
     */
    linkp=(struct linkblk *) mp->b_cont->b_rptr;
    muxbot = NULL;
    mp->b_datap->db_type = M_IOCACK;
    iocp->ioc_count = 0;
    greply(q, mp);
    break;
default:
iocnak:
    /* fail ioctl */
    mp->b_datap->db_type = M_IOCNAK;
    greply(q, mp);
}
break;
}
case M_FLUSH:
    if (*mp->b_rptr & FLUSHW)
        flushq(q, FLUSHDATA);
    if (*mp->b_rptr & FLUSHR) {
        *mp->b_rptr &= ~FLUSHW;
        greply(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;
    if (canputnext(muxbot)) {
        mblk_t *bp;
        if ((bp = allob(1, BPRI_MED)) == NULL) {
            putbq(q, mp);
            mux->bufcid = bufcall(1, BPRI_MED,
                mux_qenable, (long)q);
            break;
        }
        *bp->b_wptr++ = (struct muz *)q->ptr - mux_mux;
        bp->b_cont = mp;
    } else {
        putbq(q, mp);
    }
    break;
    default:
    bad:
        /*
        * Send an error message upstream.
        */
        mp->b_datap->db_type = M_ERROR;
        mp->b_rptr = mp->b_wptr = mp->b_datap->db_base;
        *mp->b_wptr++ = EINVAL;
        qreply(q, mp);
    }
}

```

First, there is a check to enforce that the Stream associated with minor device 0 will be the single, controlling Stream. The `ioctl`s are only accepted on this Stream. As described previously, a controlling Stream is the one that issues the `I_LINK`. Having a single control Stream is a recommended practice. `I_LINK` and `I_UNLINK` include a `linkblk` structure containing:

`l_qtop`

The upper write queue from which the `ioctl` is coming. It always equals `q` for an `I_LINK`; it is always `NULL` for `I_PLINK`.

`l_qbot`

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

`l_index`

A unique (system-wide) identifier for the link. It can be used for routing or during selective unlinks. Since the example only supports a single link, *`l_index`* is not used.

For `I_LINK`, *`l_qbot`* is saved in *`muxbot`* and a positive acknowledgment is generated. From this point on, until an `I_UNLINK` occurs, data from upper queues will be routed through *`muxbot`*. Note that when an `I_LINK` is received, the lower Stream has already been connected. This allows the driver to send messages downstream to perform any initialization functions. Returning an `M_IOCNAK` message (negative acknowledgment) in response to an `I_LINK` will cause the lower Stream to be disconnected.

The `I_UNLINK` handling code nulls out *`muxbot`* and generates a positive acknowledgment. A negative acknowledgment should not be returned to an `I_UNLINK`. The Stream head assures that the lower Stream is connected to a multiplexer before sending an `I_UNLINK M_IOCTL`.

`muxuwput` handles `M_FLUSH` messages as a normal driver would, except that there are no messages queued on the upper read queue, so there is no need to call *`flushq`* if `FLUSHR` is set.

`M_DATA` messages are not placed on the lower write message queue. They are queued on the upper write message queue. When flow control subsides on the lower Stream, the lower *`service`* procedure, *`mux/wsrv`*, is scheduled to start output. This is similar to starting output on a device driver.

Upper Write Service Procedure

The following example shows the code for the upper multiplexer write service procedure:

```
static int muxuwsrv(queue_t *q)
{
    mblk_t *mp;
    struct mux *muxe;
    muxe = (struct mux *)q->q_ptr;

    if (!muxbot) {
        flushq(q, FLUSHALL);
        return (0);
    }
}
```

```

    }
    if (muxerr) {
        flushq(q, FLUSHALL);
        return (0);
    }
    while (mp = getq(q)) {
        if (canputnext(muxbot))
            putnext(muxbot, mp);
        else {
            putbq(q, mp);
            return(0);
        }
    }
    return (0);
}

```

As long as there is a Stream still linked under the multiplexer and there are no errors, the `service` procedure will take a message off the queue and send it downstream, if flow control allows.

Lower Write Service Procedure

muxlwsrv, the lower (linked) queue write service procedure is scheduled as a result of flow control subsiding downstream (it is back-enabled).

```

static int muxlwsrv(queue_t *q)
{
    int i;

    for (i = 0; i < mux_cnt; i++)
        if (mux_mux[i].qptr && mux_mux[i].qptr->q_first)
            qenable(mux_mux[i].qptr);
    return (0);
}

```

muxlwsrv steps through all possible upper queues. If a queue is active and there are messages on the queue, then its upper write service procedure is enabled via `qenable()`.

Lower Read Put Procedure

The lower (linked) queue read put procedure is:

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

    if(muxerr) {
        freemsg(mp);
        return (0);
    }

    switch(mp->b_datap->db_type) {
    case M_FLUSH:
        /*
         * Flush queues. NOTE: sense of tests is reversed
         * since we are acting like a "stream head"
         */
        if (*mp->b_rptr & FLUSHW) {
            *mp->b_rptr &= ~FLUSHR;
            qreply(q, mp);
        } else
            freemsg(mp);
        break;
    case M_ERROR:
    case M_HANGUP:
        muxerr = 1;
        freemsg(mp);
        break;
    case M_DATA:
        /*
         * Route message. First byte indicates
         * device to send to. No flow control.
         *
         * Extract and delete device number. If the
         * leading block is now empty and more blocks
         * follow, strip the leading block.
         */
        device = *mp->b_rptr++;
    }
```

```

        /* Sanity check. Device must be in range */
        if (device < 0 || device >= mux_cnt) {
            freemsg(mp);
            break;
        }
        /*
         * If upper stream is open and not backed up,
         * send the message there, otherwise discard it.
         */
        uq = mux_mux[device].qp_ptr;
        if (uq != NULL && canputnext(uq))
            putnext(uq, mp);
        else
            freemsg(mp);
        break;
    default:
        freemsg(mp);
    }
    return (0);
}

```

muxlrput receives messages from the linked Stream. In this case, it is acting as a Stream head. It handles `M_FLUSH` messages. Note the code is reversed from that of a driver, handling `M_FLUSH` messages from upstream. There is no need to flush the read queue because no data is ever placed in it.

muxlrput also handles `M_ERROR` and `M_HANGUP` messages. If one is received, it locks-up the upper Streams by setting *muxerr*.

`M_DATA` messages are routed by looking at the first data byte of the message. This byte contains the minor device of the upper Stream. Several sanity checks are made:

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

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

The upper Stream close routine simply clears the mux entry so this queue will no longer be found. Outstanding bufcalls are not cleared.

```
/*
 * Upper queue close
 */
static int
muxclose(queue_t *q, int flag, cred_t *credp)
{
    struct mux *mux;

    mux = (struct mux *) q->q_ptr;
    qprocsoff(q);
    if (mux->bufcid != 0)
        unbufcall(mux->bufcid);
    mux->bufcid = 0;
    mux->ptr = NULL;
    q->q_ptr = NULL;
    WR(q)->q_ptr = NULL;
    return(0);
}
```

Persistent Links

With `I_LINK` and `I_UNLINK` ioctls the file descriptor associated with the Stream above the multiplexer used to set up the lower multiplexer connections must remain open for the duration of the configuration. Closing the file descriptor associated with the controlling Stream will dismantle the whole multiplexing configuration. Some applications may not want to keep a process running merely to hold the multiplexer configuration together. Therefore, “free-standing” links below a multiplexer are needed. A persistent link is such a link. It is similar to a STREAMS multiplexer link, except that a process is not needed to hold the links together. After the multiplexer has been set up, the process may close all file descriptors and exit, and the multiplexer will remain intact.

Two ioctls, `I_PLINK` and `I_PUNLINK`, are used to create and remove persistent links that are associated with the Stream above the multiplexer. `close(2)` and `I_UNLINK` are not able to disconnect the persistent links (see `strconf(1)` and `strchg(1)`).

The format of `I_PLINK` is:

```
ioctl(fd0, I_PLINK, fd1)
```

The first file descriptor, *fd0*, must reference the Stream connected to the multiplexing driver and the second file descriptor, *fd1*, must reference the Stream to be connected below the multiplexer. The persistent link can be created in the following way:

```
upper_stream_fd = open("/dev/mux", O_RDWR);
lower_stream_fd = open("/dev/driver", O_RDWR);
muxid = ioctl(upper_stream_fd, I_PLINK, lower_stream_fd);
/*
 * save muxid in a file
 */
exit(0);
```

Figure 10-8 shows how `open(2)` establishes a Stream between the device and the Stream head.

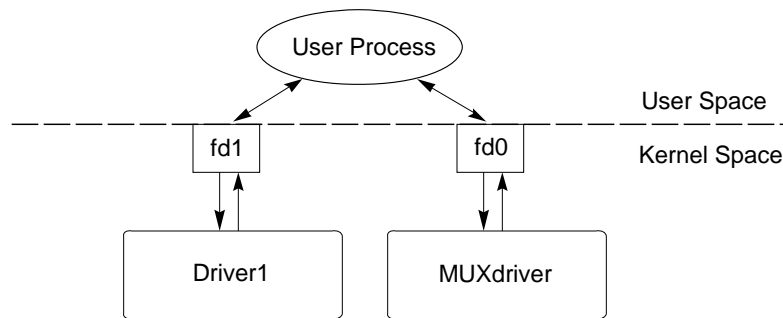


Figure 10-8 `open()` of MUXdriver and Driver1

The persistent link can still exist even if the file descriptor associated with the upper Stream to the multiplexing driver is closed. The `I_PLINK` `ioctl` returns an integer value, *muxid*, that can be used for dismantling the multiplexing configuration. If the process that created the persistent link still exists, it may

pass the *muxid* value to some other process to dismantle the link, if the dismantling is desired, or it can leave the *muxid* value in a file so that other processes may find it later. Figure 10-9 shows a multiplexer after `I_PLINK`.

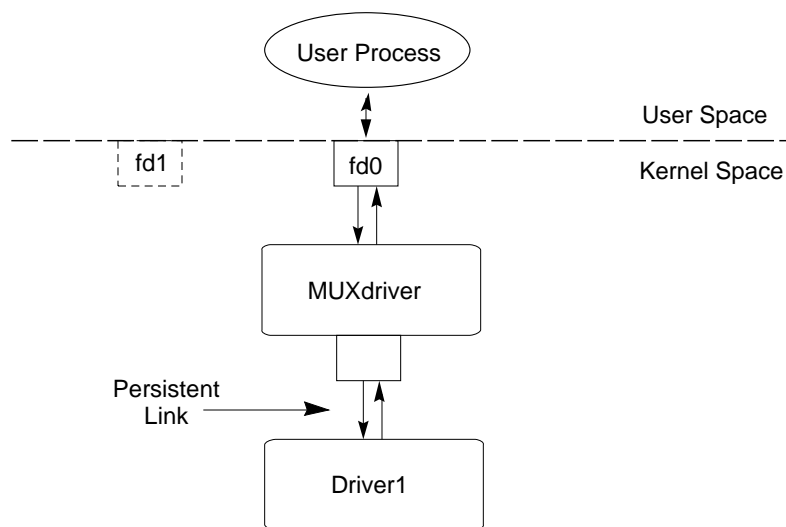


Figure 10-9 Multiplexer After `I_PLINK`

Several users can open the MUXdriver and send data to the Driver1 since the persistent link to the Driver1 remains intact. This is shown in the Figure 10-10.

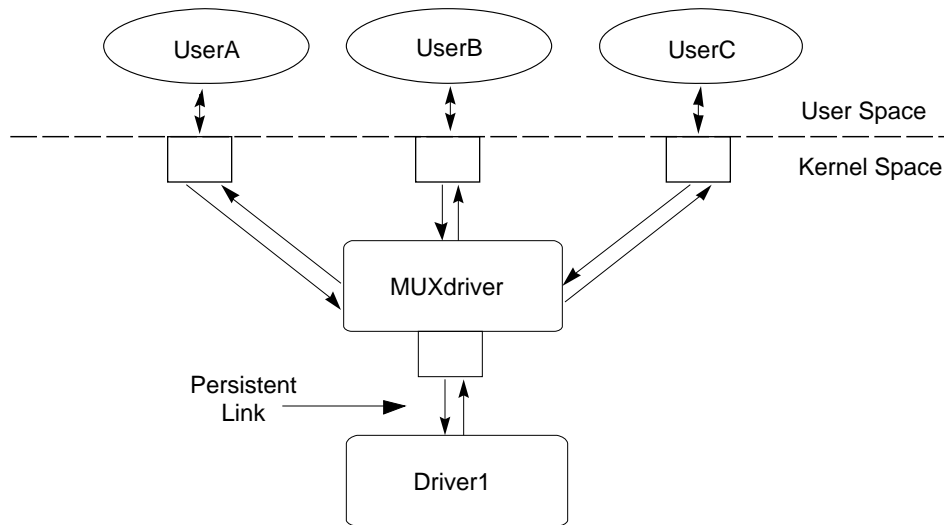


Figure 10-10 Other Users Opening a MUXdriver

The `ioctl I_PUNLINK` is used for dismantling the persistent link. Its format is:

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

where the `fd0` is the file descriptor associated with Stream connected to the multiplexing driver from above. The `muxid` is returned by the `ioctl I_PLINK` for the Stream that was connected below the multiplexer. The `I_PUNLINK` removes the persistent link between the multiplexer referenced by the `fd0` and the Stream to the driver designated by the `muxid`. Each of the bottom persistent links can be disconnected individually. An `I_PUNLINK` `ioctl` with the `muxid` value of `MUXID_ALL` will remove all persistent links below the multiplexing driver referenced by the `fd0`.

The following code example shows how to dismantle the previously given configuration:

```
fd = open("/dev/mux", O_RDWR);
/*
 * retrieve muxid from the file
 */
ioctl(fd, I_PUNLINK, muxid);
exit(0);
```

The use of the `ioctls` `I_PLINK` and `I_PUNLINK` should not be mixed with the `I_LINK` and `I_UNLINK`. Any attempt to unlink a regular link via the `I_PUNLINK` or to unlink a persistent link via the `I_UNLINK` `ioctl` will cause the *errno* value of `EINVAL` to be returned.

Since multi-level multiplexing configurations are allowed in STREAMS, it is possible to have a situation where persistent links exist below a multiplexer whose Stream is connected to the above multiplexer by regular links. Closing the file descriptor associated with the controlling Stream will remove the regular link but not the persistent links below it. On the other hand, regular links are allowed to exist below a multiplexer whose Stream is connected to the above multiplexer via persistent links. In this case, the regular links will be removed if the persistent link above is removed and no other references to the lower Streams exist.

The construction of cycles is not allowed when creating links. A cycle could be constructed by creating a persistent link of multiplexer 2 below multiplexer 1 and then closing the controlling file descriptor associated with the multiplexer 2 and reopening it again and then linking the multiplexer 1 below the multiplexer 2. This is not allowed. The operating system prevents a multiplexer configuration from containing a cycle to ensure that messages can not be routed infinitely, thus creating an infinite loop or overflowing the kernel stack.

Design Guidelines

The following lists general multiplexer design guidelines:

- The upper half of the multiplexer acts like the end of the upper Stream. The lower half of the multiplexer acts like the head of the lower Stream. Service procedures are used for flow control.

- Message routing is based on multiplexer specific criteria.
- When one Stream is being fed by many Streams, flow control may have to take place. Then all feeding Streams on the other end of the multiplexer will have to be enabled when the flow control is relieved.
- When one Stream is feeding many Streams, flow control may also have to take place. Be careful not to starve other Streams when one becomes flow controlled.

STREAMS-Based Pipes and FIFOs 11

Overview of Pipes and FIFOs

A pipe in the SunOS 5.3 system is a mechanism that provides a communication path between multiple processes. Prior to SunOS 5.0, SunOS had *standard* pipes and named pipes (also called FIFOs, or First-In-First-Out). With standard pipes, one end was opened for reading and the other end for writing, thus data flow was unidirectional. FIFOs had only one end and typically one process opened the file for reading and another process opened the file for writing. Data written into the FIFO by the writer could then be read by the reader.

To provide greater support and development flexibility for networked applications, pipes and FIFOs have become STREAMS-based in SunOS 5.3. The basic interface remains the same but the underlying implementation has changed. Pipes now provide a bidirectional mechanism for process communication. When a pipe is created via the `pipe(2)` system call, two Streams are opened and connected side-by-side, thus providing a full-duplex mechanism. Data flow is on First-In-First-Out (FIFO) basis. Previously, pipes were associated with character devices and the creation of a pipe was limited to the capacity and configuration of the device. STREAMS-based pipes and FIFOs are not attached to STREAMS-based character devices. This eliminates configuration constraints and limits the number of opened pipes to the number of file descriptors allowed for each process.

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

Creating and Opening Pipes and FIFOs

FIFOs are created via `mknod(2)` or `mkfifo(3C)`. FIFOs behave like regular file system nodes but are distinguished from other file system nodes by the `p` in the first column when the `ls -l` command is executed. Data written to the FIFO or read from the FIFO flow up and down the Stream in STREAMS buffers. Data written by one process can be read by another process.

FIFOs are opened in the same manner as other file system nodes via the `open(2)` system call. Any data written to the FIFO can be read from the same file descriptor in the First-In-First-Out manner (serial, sequentially). Modules can also be pushed on the FIFO. See `open(2)` for the restrictions that apply when opening a FIFO.

A STREAMS-based pipe is created via the `pipe(2)` system call that returns two file descriptors, `fd[0]` and `fd[1]`. Both file descriptors are opened for reading and writing. Data written to `fd[0]` becomes data read from `fd[1]` and vice versa.

Each end of the pipe has knowledge of the other end through internal data structures. Subsequent reads, writes, and closes are aware of if the other end of the pipe is open or closed. When one end of the pipe is closed, the internal data structures provide a way to access the Stream for the other end so that an `M_HANGUP` message can be sent to its Stream head.

After successful creation of a STREAMS-based pipe, 0 is returned. If `pipe(2)` is unable to create and open a STREAMS-based pipe, it will fail with `errno` set as follows:

- `EINTR` - Signal was caught while creating the Stream heads.
- `EMFILE` - Could not allocate more file descriptors for the process.
- `ENFILE` - File table has overflowed.
- `ENOMEM` - Could not allocate two `vnodes`.
- `ENOSR` - Could not allocate resources for both Stream heads.

STREAMS modules can be added to a STREAMS-based pipe with the `ioctl(2)` `I_PUSH`. A module can be pushed onto one or both ends of the pipe (see Figure 11-1). However, a pipe maintains the concept of a midpoint so that if a module is pushed onto one end of the pipe, that module cannot be popped from the other end.

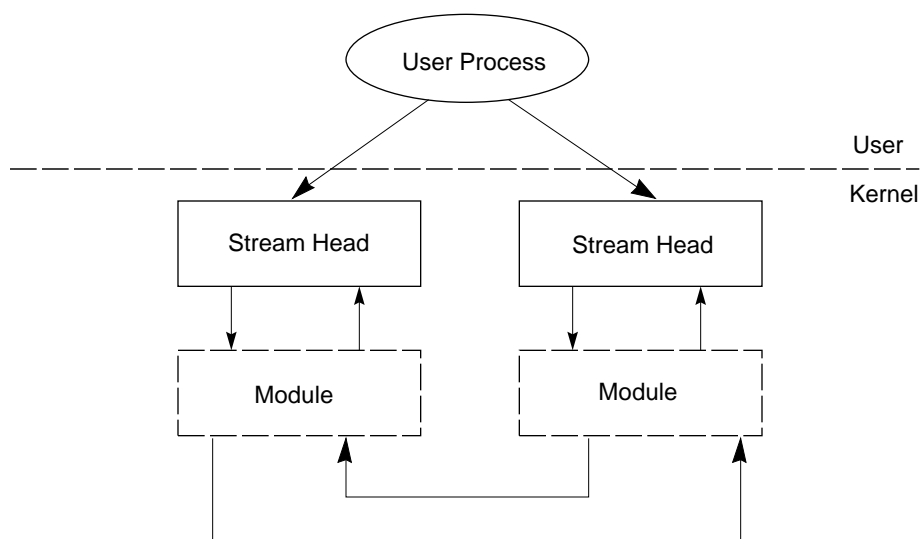


Figure 11-1 Pushing Modules on a STREAMS-based Pipe

Accessing Pipes and FIFOs

STREAMS-based pipes and FIFOs can be accessed through the operating system routines `read(2)`, `write(2)`, `ioctl(2)`, `close(2)`, `putmsg(2)`, `getmsg(2)`, and `poll(2)`. In the case of FIFOs, `open(2)` is also used.

Reading from a Pipe or FIFO

The `read(2)` (or `getmsg(2)`) system call is used to read from a pipe or FIFO. Data can be read from either end of a pipe.

On success, the `read` returns the number of bytes read and placed in the buffer. When the end of the data is reached, the `read` returns 0.

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

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

Writing to a Pipe or FIFO

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

Zero-Length Writes

If a user process issues `write(2)` with 0 as the number of bytes to send a STREAMS-based pipe or FIFO, 0 is returned, and by default no message is sent down the Stream. However, if a user must send a zero-length message downstream, an `ioctl` call may be used to change this default behavior. The flag `SNZZERO` supports this. If `SNZZERO` is set in the Stream head, `write(2)` requests of 0 bytes will generate a zero-length message and send the message down the Stream. If `SNZZERO` is not set, no message is generated and 0 is returned to the user.

The `SNZZERO` bit may be manipulated via the `ioctl I_SWROPT`. If the arg in the `ioctl` call has `SNZZERO` set, the bit is turned on. If the arg is set to 0, the `SNZZERO` bit is turned off.

The `ioctl I_GWROPT` is used to return the current write settings.

Atomic Writes

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

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

If the module packet size is at least the size of `PIPE_BUF`, the Stream head packages the data in such a way that the first message is at least `PIPE_BUF` bytes. The remaining data may be packaged into smaller or larger blocks depending on buffer availability. If the first module on the Stream cannot support a packet of `PIPE_BUF`, atomic writes on the pipe cannot be guaranteed.

Closing a Pipe or FIFO

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

Flushing Pipes and FIFOs

When the flush request is initiated from a user `ioctl` or from a `flushq()` routine, the `FLUSHR` and/or the `FLUSHW` bits of an `M_FLUSH` message must be switched. The point of switching the bits is the point where the `M_FLUSH` message is passed from a write queue to a read queue. This point is also known as the mid-point of the pipe.

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

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

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

The `pipemod` module can be pushed on any Stream if it requires the bit switching.

Named Streams

It may be necessary for some applications to associate a Stream or STREAMS-based pipe with an existing node in the file system name space. For example, a server process may create a pipe, name one end of the pipe, and allow unrelated processes to communicate with it over that named end.

fattach

A STREAMS file descriptor can be named by attaching that file descriptor to a node in the file system name space. The routine `fattach()` (see also `fattach(3C)`) is used to name a STREAMS file descriptor. Its format is:

```
int fattach (int fildes, char *path)
```

where *fildes* is an open file descriptor that refers to either a STREAMS-based pipe or a STREAMS device driver (or a pseudo device driver), and *path* is an existing node in the file system name space (for example, a regular file, directory, character special file, etc).

The *path* cannot have a Stream already attached to it. It cannot be a mount point for a file system nor the root of a file system. A user must be an owner of the *path* with write permission or a user with the appropriate privileges in order to attach the file descriptor.

If the *path* is in use when the routine `fattach()` is executed, those processes accessing the *path* will not be interrupted and any data associated with the *path* before the call to the `fattach()` routine will continue to be accessible by those processes.

After a Stream is named, all subsequent operations (for example, `open(2)`) on the *path* will operate on the named Stream. Thus, it is possible that a user process has one file descriptor pointing to the data originally associated with the *path* and another file descriptor pointing to a named Stream.

Once the Stream has been named, the `stat(2)` system call on *path* will show information for the Stream. If the named Stream is a pipe, the `stat(2)` information will show that *path* is a pipe. If the Stream is a device driver or a pseudo device driver, *path* appears as a device. The initial modes, permissions, and ownership of the named Stream are taken from the attributes of the *path*. The user can issue the system calls `chmod(2)` and `chown(2)` to alter the attributes of the named Stream and not affect the original attributes of the *path* nor the original attributes of the STREAMS file.

The size represented in the `stat(2)` information will reflect the number of unread bytes of data currently at the Stream head. This size is not necessarily the number of bytes written to the Stream.

A STREAMS-based file descriptor can be attached to many different *paths* at the same time (that is, a Stream can have many names attached to it). The modes, ownership, and permissions of these *paths* may vary, but operations on any of these *paths* will access the same Stream.

Named Streams can have modules pushed on them, be polled, be passed as file descriptors, and be used for any other STREAMS operation.

fdetach

A named Stream can be disassociated from a filename with the `fdetach()` routine (see also `fdetach(3C)`) that has the following format:

```
int fdetach (char *path)
```

where *path* is the name of the previously named Stream. Only the owner of *path* or the user with the appropriate privileges may disassociate the Stream from its name. The Stream may be disassociated from its name while processes are accessing it. If these processes have the named Stream open at the time of the `fdetach()` call, the processes will not get an error, and will continue to access the Stream. However, after the disassociation, subsequent operations on *path* access the underlying file rather than the named Stream.

If only one end of the pipe is named, the last close of the other end will cause the named end to be automatically detached. If the named Stream is a device and not a pipe, the last close will not cause the Stream to be detached.

If there is no named Stream or the user does not have access permissions on *path* or on the named Stream, `fdetach()` returns -1 with *errno* set to `EINVAL`. Otherwise, `fdetach()` returns 0 for success.

A Stream will remain attached with or without an active server process. If a server aborted, the only way a named Stream is cleaned up is if the server executed a `clean up` routine that explicitly detached and closed down the Stream.

If the named Stream is that of a pipe with only one end attached, clean up will occur automatically. The named end of the pipe is forced to be detached when the other end closes down. If there are no other references after the pipe is detached, the Stream is deallocated and cleaned up. Thus, a forced detach of a pipe end will occur when the server is aborted.

If the both ends of the pipe are named, the pipe remains attached even after all processes have exited. In order for the pipe to become detached, a server process would have to explicitly call a program that executed the `fdetach()` routine.

To eliminate the need for the server process to invoke the program, the `fdetach(1M)` command can be used. This command accepts a path name that is a path to a named Stream. When the command is invoked, the Stream is detached from the path. If the name was the only reference to the Stream, the Stream is also deallocated.

A user invoking the `fdetach(1M)` command must be an owner of the named Stream or a user with the appropriate permissions.

isastream

The function `isastream()` (see `isastream(3C)`) may be used to determine if a file descriptor is associated with a STREAM. Its format is:

```
int isastream (int fildes);
```

where *fildes* refers to an open file. `isastream()` returns 1 if *fildes* represents a STREAMS file, and 0 if not. On failure, `isastream()` returns -1 with *errno* set to `EBADF`.

This function is useful for client processes communicating with a server process over a named Stream to check whether the file has been overlaid by a Stream before sending any data over the file.

Passing File Descriptors

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

Named Streams in A Remote Environment

If a user on the server machine creates a pipe and mounts it over a file that is part of an RFS (Remote File System) advertised resource, a user on the client machine (that has remotely named the resource) may access the remote named Stream. A user on the client machine is not allowed to pass file descriptors across the named Stream and will get an error when the `ioctl` request is attempted. If a user on the client machine creates a pipe and attempts to attach it to a file that is a remotely named resource, the system call will fail.

The following three examples are given as illustrations:

- Suppose the server advertised a resource `/dev/foo`, created a STREAMS-based pipe, and attached one end of the pipe onto `/dev/foo/spipe`. All processes on the server machine will be able to access the pipe when they open `/dev/foo/spipe`. Now suppose that client XYZ mounts the advertised resource `/dev/foo` onto its `/mnt` directory. All processes on client XYZ will be able to access the STREAMS-based pipe when they open `/mnt/spipe`.
- If the server advertised another resource `/dev/fog` and client XYZ mounts that resource onto its `/install` directory and then attaches a STREAMS-based pipe onto `/install`, the mount would fail with `errno` set to `EBUSY`, because `/install` is already a mount point. If client XYZ attached a pipe onto `/install/spipe`, the mount would also fail with `errno` set to `EREMOTE`, because the mount would require crossing an RFS mount point.
- Suppose the server advertised its `/usr/control` directory and client XYZ mounts that resource onto its `/tmp` directory. The server now creates a STREAMS-based pipe and attaches one end over its `/usr` directory. When the server opens `/usr` it will access the pipe. On the other hand, when the client opens `/tmp` it will access what is in the server's `/usr/control` directory.

Unique Connections

With named pipes, client processes may communicate with a server process via a module called `connld` that enables a client process to gain a unique, non-multiplexed connection to a server. The `connld` module can be pushed onto the named end of the pipe. If `connld` is pushed on the named end of the pipe and that end is opened by a client, a new pipe will be created. One file descriptor for the new pipe is passed back to a client (named Stream) as the file

descriptor from the `open(2)` system call and the other file descriptor is passed to the server via `ioctl I_RECUFD`. The server and the client may now communicate through a new pipe.

Figure 11-2 illustrates a server process that has created a pipe and pushed the `connld` module on the other end. The server then invokes the `fattach()` routine to name the other end `/usr/toserv`.

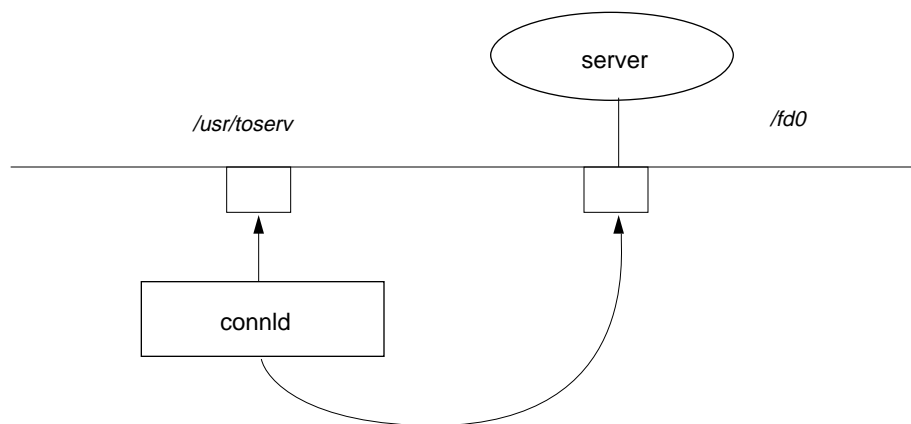


Figure 11-2 Server Sets Up a Pipe

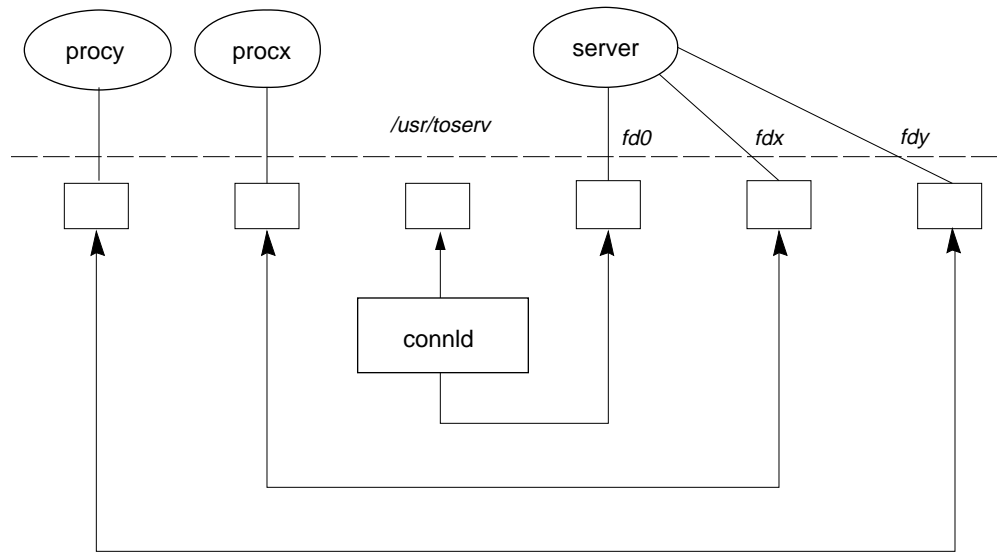


Figure 11-3 Processes X and Y Open */usr/toserv*

When process X (procx) opens */usr/toserv*, it gains a unique connection to the server process that was at one end of the original STREAMS-based pipe. When process Y (procy) does the same, it also gains a unique connection to the server. As shown in Figure 11-3, the server process has access to three separate STREAMS-based pipes via three file descriptors.

`connld` is a STREAMS-based module that has an `open`, `close`, and `put` procedure.


When the named Stream is opened, the `open` routine of `connld` is called. The `connld` `open` will fail if:

- The pipe ends can not be created.
- A file pointer and file descriptor can not be allocated.
- The Stream head can not stream the two pipe ends.
- `strioc10` fails while sending the file descriptor to the server.

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

The `connld` module does not process messages. All messages are passed to the next object in the Stream. The `read` and `write` routines call `putnext()` to send the message up or down the Stream.

STREAMS-Based Terminal Subsystem

12 

Overview of Terminal Subsystem

STREAMS provides a uniform interface for implementing character I/O devices and networking protocols in the kernel. SunOS 5.3 implements the terminal subsystem in STREAMS. The STREAMS-based terminal subsystem (Figure 12-1) provides many benefits:

- Reusable line discipline modules. The same module can be used in many STREAMS where the configuration of these STREAMS may be different.
- Line-discipline substitution. Although Sun provides a standard terminal line-discipline module, another one conforming to the interface can be substituted. For example, a remote login feature may use the terminal subsystem line discipline module to provide a terminal interface to the user.
- Internationalization. The modularity and flexibility of the STREAMS-based terminal subsystem enables an easy implementation of a system that supports multiple-byte characters for internationalization. This modularity also allows easy addition of new features to the terminal subsystem.
- Easy customizing. Users may customize their terminal subsystem environment by adding and removing modules of their choice.
- The pseudo-terminal subsystem. The pseudo-terminal subsystem can be easily supported (this is discussed in more detail in the section “STREAMS-based Pseudo-Terminal Subsystem” later in this chapter).
- Merge with networking. By pushing a line discipline module on a network line, you can make the network look like a terminal line.

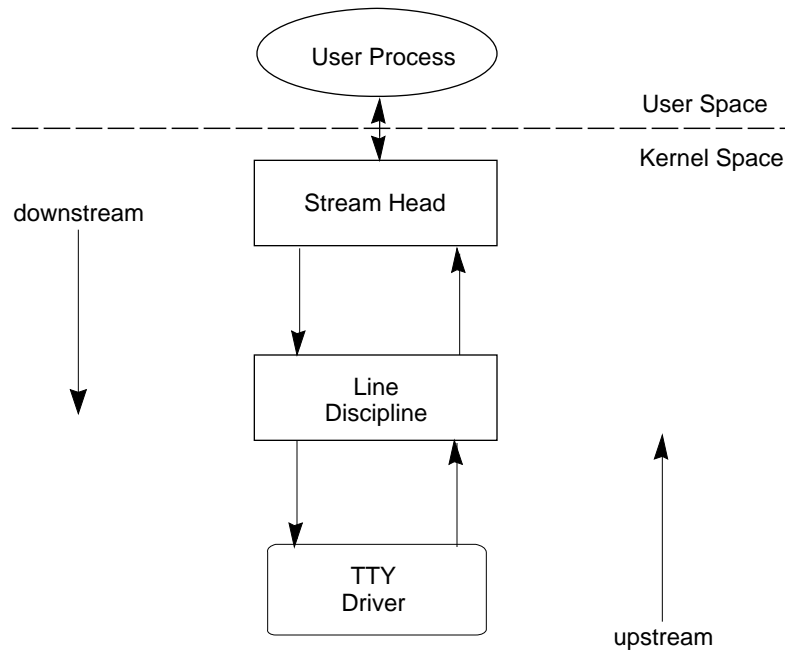


Figure 12-1 STREAMS-based Terminal Subsystem

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

The STREAMS-based terminal subsystem supports `termio(7)`, the `termios(3)` specification of the POSIX standard, multiple byte characters for internationalization, the interface to asynchronous hardware flow control (see `termiox(7)`), and peripheral controllers for asynchronous terminals. XENIX® and BSD compatibility can also be provided by pushing the `ttcompat` module.

To use `sh1` with the STREAMS-based terminal subsystem, the `sxt` driver is implemented as a STREAMS-based driver. However, the `sxt` feature is being discontinued and users are encouraged to use the job-control mechanism. Note that both `sh1` and job control should *not* be run simultaneously.

Line-Discipline Module

A STREAMS line-discipline module called `ldterm` (see `ldterm(7)`) is a key part of the STREAMS-based terminal subsystem. Throughout this chapter, the terms *line discipline* and `ldterm` are used interchangeably and refer to the STREAMS version of the standard line discipline and not the traditional character version. `ldterm` performs the standard terminal I/O processing that was traditionally done through the *linesw* mechanism.

The `termio` and `termios` specifications describe four flags that are used to control the terminal:

- `c_iflag` (defines input modes)
- `c_oflag` (defines output modes)
- `c_cflag` (defines hardware control modes)
- `c_lflag` (defines terminal functions used by `ldterm`).

To process these flags elsewhere (for example, in the firmware or in another process), a mechanism is in place to turn on and off the processing of these flags. When `ldterm` is pushed, it sends an `M_CTL` message downstream that asks the driver which flags the driver will process. The driver sends back that message in response if it needs to change `ldterm`'s default processing. By default, `ldterm` assumes that it must process all flags except `c_cflag`, unless it receives a message telling otherwise.

Default Settings

When `ldterm` is pushed on the Stream, the `open` routine initializes the settings of the `termio` flags. The default settings are:

```
c_iflag = BRKINT | ICRNL | IXON | IMAXBEL
```

```
c_oflag = OPOST | ONLCR | TAB3
```

```
c_cflag = CREAD | CS8 | B9600
```

```
c_lflag = ISIG | ICANON | ECHO | ECHOK | IEXTEN | ECHOE | ECHOKE | ECHOCTL
```

In canonical mode (`ICANON` flag in `c_lflag` is turned on), read from the terminal file descriptor is in message non-discard (`RMSGN`) mode (see `streamio(7)`). This implies that in canonical mode, read on the terminal file

descriptor always returns at most one line regardless how many characters have been requested. In non-canonical mode, `read` is in byte-stream (`RNORM`) mode. The flag `ECHOCTL` has been added for SunOS 4.1 compatibility.

User-Configurable Settings

See `termio(7)` for more information.

Open and Close Routines

The `open` routine of the `ldterm` module allocates space for holding the TTY structure (see `ldtermstd_state_t` in `ldterm.h`) by allocating a buffer from the STREAMS buffer pool. The number of modules that can be pushed on one stream, as well as the number of TTY's in use, is limited. The number of instances of `ldterm` that have been pushed is limited only by available memory. The `open` also sends an `M_SETOPTS` message upstream to set the Stream head high and low water marks to 1024 and 200, respectively. These are the current values (although they may change over time).

The `ldterm` module identifies itself as a TTY to the stream head by sending an `M_SETOPTS` message upstream with the `SO_ISTTY` bit of `so_flags` set. The Stream head allocates the controlling TTY on the `open`, if one is not already allocated.

To maintain compatibility with existing applications that use the `O_NDELAY` flag, the `open` routine sets the `SO_NDELOK` flag on in the `so_flags` field of the `stroptions` structure in the `M_SETOPTS` message.

The `open` routine fails if there are no buffers available (cannot allocate the internal state structure) or when an interrupt occurs while waiting for a buffer to become available.

The `close` routine frees all the outstanding buffers allocated by this Stream. It also sends an `M_SETOPTS` message to the Stream head to undo the changes made by the `open` routine. The `ldterm` module also sends `M_START` messages downstream to undo the effect of any previous `M_STOP` messages.

Read-Side Processing

The `ldterm` module's read side processing has `put` and `service` procedures. High and low water marks for the read queue are 1024 and 200, respectively. These are the current values and may be subject to change.

`ldterm` can send the following messages upstream:

`M_DATA`, `M_BREAK`, `M_PCSIG`, `M_SIG`, `M_FLUSH`, `M_ERROR`, `M_IOCACK`, `M_IOCNAK`, `M_HANGUP`, `M_CTL`, `M_SETOPTS`, `M_COPYOUT`, and `M_COPYIN`.

The `ldterm` module's read side processes `M_BREAK`, `M_DATA`, `M_CTL`, `M_FLUSH`, `M_HANGUP`, `M_IOCACK` and `M_IOCNAK` messages. All other messages are sent upstream unchanged.

The `put` procedure scans the message for flow-control characters (IXON), signal-generating characters, and after (possible) transformation of the message, queues the message for the `service` procedure. Echoing is handled completely by the `service` procedure.

In canonical mode if the `ICANON` flag is on in `c_lflag`, canonical processing is performed. If the `ICANON` flag is off, non-canonical processing is performed (see `termio(7)` for more details). Handling of `VMIN/VTIME` in the STREAMS environment is somewhat complicated, because `read` needs to activate a timer in the `ldterm` module in some cases; hence, read notification becomes necessary. When a user issues an `ioctl` to put `ldterm` in non-canonical mode, the `ldterm` module sends an `M_SETOPTS` message to the Stream head to register read notification. Further reads on the terminal file descriptor will cause the Stream head to issue an `M_READ` message downstream and data will be sent upstream in response to the `M_READ` message. With read notification, buffering of raw data is performed by `ldterm`. It is possible to canonize the raw data, when the user has switched from raw to canonical mode. However, the reverse is not possible.

To summarize, in non-canonical mode, the `ldterm` module buffers all data until `VMIN` or `VTIME` criteria are met. For example, if `VMIN=3` and `VTIME=0`, and three bytes have been buffered by `ldterm`, these characters would be sent to the stream head *regardless* of whether there is a pending `M_READ`, and no `M_READ` would need to be sent down. If an `M_READ` message is received, the number of bytes sent upstream will be the argument of the `M_READ` message, unless `VTIME` is satisfied before `VMIN` (for example, the timer has expired) in which case whatever characters are available will be sent upstream.

The service procedure of `ldterm` handles STREAMS related flow control. Since the read side high and low water marks are 1024 and 200 respectively, placing 1024 characters or more on the `ldterm`'s read queue will cause the `QFULL` flag be turned on indicating that the module below should not send more data upstream.

Input flow control is regulated by the line-discipline module by generating `M_STARTI` and `M_STOPI` high priority messages. When sent downstream, receiving drivers or modules take appropriate action to regulate the sending of data upstream. Output flow control is activated when `ldterm` receives flow control characters in its data stream. The `ldterm` module then sets an internal flag indicating that output processing is to be restarted/stopped and sends an `M_START/M_STOP` message downstream.

Write-Side Processing

Write side processing of the `ldterm` module is performed by the write-side `put` and `service` procedures.

The `ldterm` module supports the following `ioctls`:

`TCSETA`, `TCSETAW`, `TCSETAF`, `TCSETS`, `TCSETSW`, `TCSETSF`, `TCGETA`, `TCGETS`, `TCXONC`, `TCFLSH` and `TCSBRK`.

All `ioctls` not recognized by the `ldterm` module are passed downstream to the neighboring module or driver.

The following messages can be received on the write side:

`M_DATA`, `M_DELAY`, `M_BREAK`, `M_FLUSH`, `M_STOP`, `M_START`, `M_STOPI`, `M_STARTI`, `M_READ`, `M_IOCDATA`, `M_CTL`, and `M_IOCTL`.

On the write side, the `ldterm` module processes `M_FLUSH`, `M_DATA`, `M_IOCTL`, and `M_READ` messages, and all other message are passed downstream unchanged.

An `M_CTL` message is generated by `ldterm` as a query to the driver for an intelligent peripheral and to decide on the functional split for `termio` processing. If all or part of `termio` processing is done by the intelligent peripheral, `ldterm` can turn off this processing to avoid computational overhead. This is done by sending an appropriate response to the `M_CTL` message, as follows: (see also `ldterm(7)`).

- If all of the `termio` processing is done by the peripheral hardware, the driver sends an `M_CTL` message back to `ldterm` with `ioc_cmd` of the structure `iocblk` set to `MC_NO_CANON`. If `ldterm` is to handle all `termio` processing, the driver sends an `M_CTL` message with `ioc_cmd` set to `MC_DO_CANON`. The default is `MC_DO_CANON`.
- If the peripheral hardware handles only part of the `termio` processing, it informs `ldterm` in the following way:

The driver for the peripheral device allocates an `M_DATA` message large enough to hold a `termios` structure. The driver then turns on those `c_iflag`, `c_oflag`, and `c_lflag` fields of the `termios` structure that are processed on the peripheral device by ORing the flag values. The `M_DATA` message is then attached to the `b_cont` field of the `M_CTL` message it received. The message is sent back to `ldterm` with `ioc_cmd` in the data buffer of the `M_CTL` message set to `MC_PART_CANON`.

One difference between AT&T STREAMS and SunOS 5.x is that AT&T's line discipline module does not check if write side flow control is in effect before forwarding data downstream. It expects the downstream module or driver to add the messages to its queue until flow control is lifted. This is not true in SunOS 5.x.

EUC Handling in ldterm

The idea of letting post-processing (the `o_flags`) happen off the host processor is not recommended unless the board software is prepared to deal with international (EUC) character sets properly. The reason for this is that post-processing must take the EUC information into account. `ldterm` knows about the screen width of characters (that is, how many columns are taken by characters from each given code set on the current physical display) and it takes this width into account when calculating tab expansions. When using multi-byte characters or multi-column characters `ldterm` automatically handles tab expansion (when `TAB3` is set) and does not leave this handling to a lower module or driver.

By default, multi-byte handling by `ldterm` is turned off. When `ldterm` receives an `EUC_WSET` `ioctl` call, it turns multi-byte processing on, if it is essential to properly handle the indicated code set. Thus, if one is using single byte 8-bit codes and has no special multi-column requirements, the special

multi-column processing is not used at all. This means that multi-byte processing does not reduce the processing speed or efficiency of `ldterm` unless it is actually used.

The following describes how the EUC handling in `ldterm` works:

First, the multi-byte and multi-column character handling is only enabled when the `EUC_WSET` `ioctl` indicates that one of the following conditions is met:

- Code set consists of more than one byte (including the SS2 and/or SS3) of characters
- Code set requires more than one column to display on the current device, as indicated in the `EUC_WSET` structure.

Assuming that one or more of the a previous conditions, EUC handling is enabled. At this point, a parallel array (see `ldterm_mod` structure) used for other information, is allocated and a pointer to it is stored in `t_eucp_mp`. The parallel array that it holds is pointed to by `t_eucp`. The `t_codeset` field holds the flag that indicates which of the code sets is currently being processed on the read side. When a byte with the high bit arrives, it is checked to see if it is SS2 or SS3. If so, it belongs to code set 2 or 3. Otherwise, it is a byte that comes from code set 1. Once the extended code set flag has been set, the input processor retrieves the subsequent bytes, as they arrive, to build one multi-byte character. The counter field `t_eucleft` tells the input processor how many bytes remain to be read for the current character. The parallel array `t_eucp` holds for each logical character in the canonical buffer its display width. During erase processing, positions in the parallel array are consulted to determine how many backspaces need to send to erase each logical character. (In canonical mode, one backspace of input erases one logical character, no matter how many bytes or columns that character consumes.) This greatly simplifies erase processing for EUC.

The `t_maxeuc` field holds the maximum length, in memory bytes, of the EUC character mapping currently in use. The `eucwioc` field is a sub-structure that holds information about each extended code set.

The `t_eucign` field aids in output post-processing (tab expansion). When characters are output, `ldterm` keeps a column to indicate what the current cursor column is supposed to be. When it sends the first byte of an extended character, it adds the number of columns required for that character to the output column. It then subtracts one from the total width in memory bytes of

that character and stores the result in *t_eucign*. This field tells *ldterm* how many subsequent bytes to ignore for the purposes of column calculation. (*ldterm* calculates the appropriate number of columns when it sees the first byte of the character.)

The field *t_eucwarn* is a counter for occurrences of bad extended characters. It is mostly useful for debugging. After receiving a certain number of illegal EUC characters (perhaps because of some problem on the line or with declared values), a warning is given on the system console.

There are two relevant files for handling multi-byte characters: *euc.h* and *eucontrol.h*. The *eucontrol.h* contains the structure that is passed with *EUC_WSET* and *EUC_WGET* calls. The normal way to use this structure is to get *CSWIDTH* from the *locale* via a mechanism such as *getwidth* or *setlocale* and then copy the values into the structure in *eucontrol.h*, and send the structure via an *I_STR ioctl* call. The *EUC_WSET* call informs the *ldterm* module about the number of bytes in extended characters and how many columns the extended characters from each set consume on the screen. This allows *ldterm* to treat multi-byte characters as single units for the purpose of erase processing and to correctly calculate tab expansions for multi-byte characters.

Note – *LC_CTYPE* (instead of *CSWIDTH*) should be used in the environment in SunOS 5.x systems. See *chrtbl(1M)* for more information.

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

Code Example 12-1 EUC

```
#include <eucontrol.h>          /* need others, like stropts.h */

struct eucontrol eucw;          /*for EUC_WSET/WGET to line disc*/
eucwidth_t width;               /* ret struct from _getwidth() */
/*
 * set_euc          Send EUC code widths to line discipline.
 */
set_euc(struct eucontrol *e)
{
```

Code Example 12-1 EUC

```

    struct strioctl sb;

    sb.ic_cmd = EUC_WSET;
    sb.ic_timeout = 15;
    sb.ic_len = sizeof(struct eucioc);
    sb.ic_dp = (char *) e;

    if (ioctl(0, I_STR, &sb) < 0)
        fail();
}
/*
 * euclook.    Get current EUC code widths from line discipline.
 */
euclook(struct eucioc *e)
{
    struct strioctl sb;

    sb.ic_cmd = EUC_WGET;
    sb.ic_timeout = 15;
    sb.ic_len = sizeof(struct eucioc);
    sb.ic_dp = (char *) e;

    if (ioctl(0, I_STR, &sb) < 0)
        fail();

    printf("CSWIDTH=%d:%d,%d:%d,%d:%d",
           e->eucw[1], e->scrw[1],
           e->eucw[2], e->scrw[2],
           e->eucw[3], e->scrw[3]);
}

```

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

Hardware Emulation Module

If a Stream supports a terminal interface, a driver or module that understands all `ioctl`s to support terminal semantics (specified by `termio` and `termios`) is needed. If there is no hardware driver that understands all `ioctl` commands downstream from the `ldterm` module, a hardware emulation module must be placed downstream from the line discipline module. The

function of the hardware emulation module is to understand and acknowledge the `ioctl`s that may be sent to the process at the Stream head and to mediate the passage of control information downstream. The combination of the line-discipline module and the hardware emulation module behaves as if there were an actual terminal on that Stream.

The hardware emulation module is necessary whenever there is no TTY driver at the end of the Stream. For example, the module is necessary in a pseudo-TTY situation where there is process- to- process communication on one system (this is discussed later in this chapter), or in a network situation where a `termio` interface is expected (for example, remote login) but there is no TTY driver on the Stream.

Most of the actions taken by the hardware emulation module are the same regardless of the underlying architecture. However, there are some actions that are different depending on whether the communication is local or remote and whether the underlying transport protocol is used to support the remote connection.

Each hardware emulation module has an `open`, `close`, `read queue put` procedure, and `write queue put` procedure.

The hardware emulation module does the following:

- Processes, if appropriate, and acknowledges receipt of the following `ioctl`s on its write queue by sending an `M_IOCACK` message back upstream: `TCSETA`, `TCSETAW`, `TCSETAF`, `TCSETS`, `TCSETSW`, `TCSETSF`, `TCGETA`, `TCGETS`, and `TCSBRK`.
- Acknowledges the Extended UNIX Code (EUC) `ioctl`s.
- If the environment supports windowing, it acknowledges the windowing `ioctl`s `TIOCSWINSZ`, `TIOCGWINSZ`, and `JWINSIZE`. If the environment does not support windowing, an `M_IOCNAK` message is sent upstream.
- If any other `ioctl`s are received on its write queue, it sends an `M_IOCNAK` message upstream. It doesn't pass any unrecognized `ioctl`'s to the slave driver.
- When the hardware emulation module receives an `M_IOCTL` message of type `TCSBRK` on its write queue, it sends an `M_IOCACK` message upstream and the appropriate message downstream. For example, an `M_BREAK` message could be sent downstream.

- When the hardware emulation module receives an `M_IOCTL` message on its write queue to set the baud rate to 0 (`TCSETAW` with `CBAUD` set to `B0`), it sends an `M_IOCACK` message upstream and an appropriate message downstream; for networking situations this will probably be an `M_PROTO` message which is a `TPI T_DISCON_REQ` message requesting the transport provider to disconnect.
- All other messages (`M_DATA`, for instance) not mentioned here are passed to the next module or driver in the Stream.

The hardware emulation module processes messages in a way consistent with the driver that exists .

STREAMS-based Pseudo-Terminal Subsystem

The STREAMS-based pseudo-terminal subsystem provides the user with an interface that is identical to the STREAMS-based terminal subsystem described earlier in this chapter. The pseudo-terminal subsystem (pseudo-TTY) supports a pair of STREAMS-based devices called the *master* device and *slave* device. The *slave* device provides processes with an interface that is identical to the terminal interface. However, where all devices, which provide the terminal interface, have some kind of hardware device behind them, the *slave* device has another process manipulating it through the master half of the pseudo terminal. Anything written on the *master* device is given to the *slave* as an input and anything written on the *slave* device is presented as an input on the master side.

Figure 12-2 illustrates the architecture of the STREAMS-based pseudo-terminal subsystem. The *master* driver called `ptm` is accessed through the clone driver (see `clone(7)`) and is the controlling part of the system. The *slave* driver called `pts` works with the line discipline module and the hardware emulation module to provide a terminal interface to the user process. An optional packetizing module called `pckt` is also provided. It can be pushed on the master side to support `packet mode` (this is discussed later).

The number of pseudo-TTY devices that can be installed on a system depends on available memory.

Line-Discipline Module

In the pseudo-TTY subsystem, the line discipline module is pushed on the slave side to present the user with the terminal interface.

`ldterm` may turn off the processing of the `c_iflag`, `c_oflag`, and `c_lflag` fields to allow processing to take place elsewhere. The `ldterm` module can also turn off all canonical processing when it receives an `M_CTL` message with the `MC_NO_CANON` command to support *remote mode*. Although `ldterm` passes through messages without processing them, the appropriate flags are set when a *get* `ioctl`, such as `TCGETA` or `TCGETS`, is issued to indicate that canonical processing is being performed.

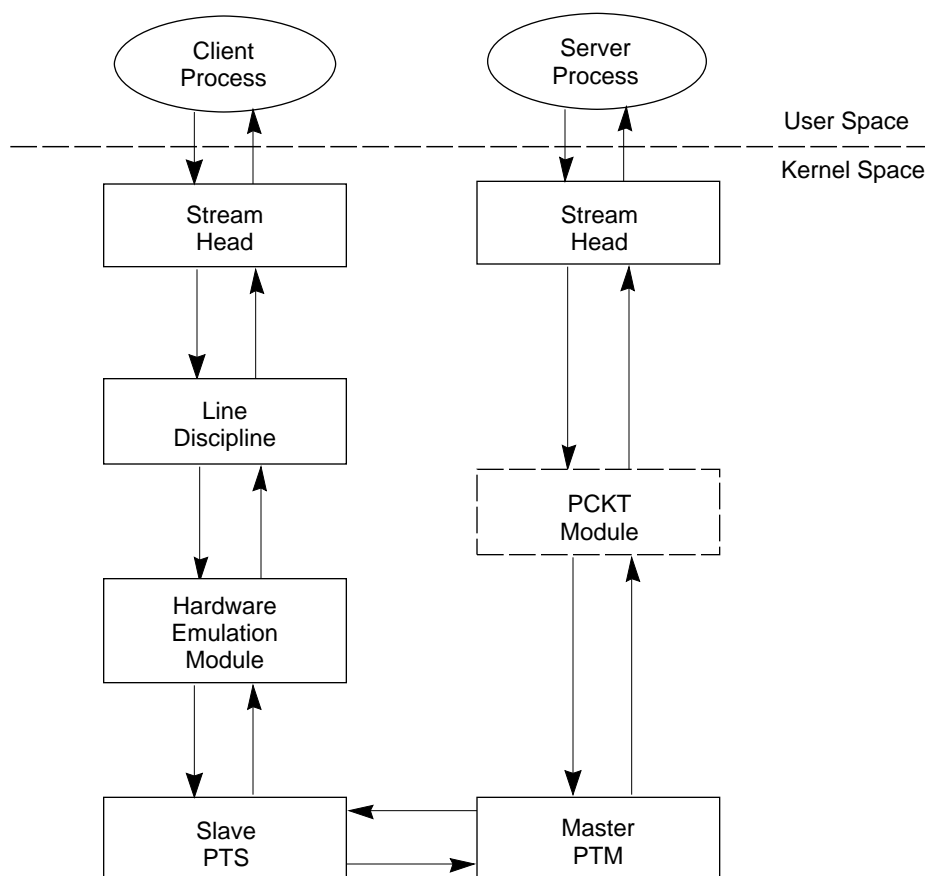


Figure 12-2 Pseudo-TTY Subsystem Architecture

Pseudo-TTY Emulation Module - PTEM

Since the pseudo-TTY subsystem has no hardware driver downstream from the `ldterm` module to process the terminal `ioctl` calls, another module that understands the `ioctl` commands is placed downstream from the `ldterm`. This module, known as `ptem`, processes all of the terminal `ioctl` commands and mediates the passage of control information downstream.

`ldterm` and `ptem` together behave like a real terminal. Since there is no real terminal or modem in the pseudo-TTY subsystem, some of the `ioctl` commands are ignored and cause only an acknowledgment of the command. The `ptem` module keeps track of the terminal parameters set by the various `set` commands such as `TCSETA` or `TCSETAW` but does not usually perform any action. For example, if one of the "set" `ioctl`s is called, none of the bits in the `c_cflag` field of `termio` has any effect on the pseudo-terminal except if the baud rate is set to 0. When setting the baud rate to 0, it has the effect of hanging up the pseudo-terminal.

The pseudo-terminal has no concept of parity so none of the flags in the `c_iflag` that control the processing of parity errors have any effect. The delays specified in the `c_oflag` field are not also supported.

The `ptem` module does the following:

- Processes, if appropriate, and acknowledges receipt of the following `ioctl`s on its write queue by sending an `M_IOCACK` message back upstream:

`TCSETA`, `TCSETAW`, `TCSETAF`, `TCSETS`, `TCSETSW`, `TCSETSF`, `TCGETA`, `TCGETS`, and `TCSBRK`.

- Keeps track of the window size; information needed for the `TIOCSWINSZ`, `TIOCGWINSZ`, and `JWINSIZE` `ioctl` commands.
- When it receives any other `ioctl` on its write queue, it sends an `M_IOCNAK` message upstream.
- It passes downstream the following `ioctl`s after processing them:

`TCSETA`, `TCSETAW`, `TCSETAF`, `TCSETS`, `TCSETSW`, `TCSETSF`, `TCSBRK`, and `TIOCSWINSZ`.

- `ptem` frees any `M_IOCNAK` messages it receives on its read queue in case the `pckt` module (`pckt` is described in the section "Packet Mode") is not on the pseudo terminal subsystem and the above `ioctl`s get to the *master's* Stream head which would then send an `M_IOCNAK` message.
- In its open routine, the `ptem` module sends an `M_SETOPTS` message upstream requesting allocation of a controlling TTY.
- When the `ptem` module receives an `M_IOCTL` message of type `TCSBRK` on its read queue, it sends an `M_IOCACK` message downstream and an `M_BREAK` message upstream.

- When the `ptem` receives an `ioctl` message on its write queue to set the baud rate to 0 (`TCSETAW` with `CBAUD` set to `B0`), it sends an `M_IOCACK` message upstream and a zero-length message downstream.
- When it receives an `M_IOCTL` of type `TIOCSIGNAL` on its read queue, it sends an `M_IOCACK` downstream and an `M_PCSIG` upstream where the signal number is the same as in the `M_IOCTL` message.
- When the `ptem` module receives an `M_IOCTL` of type `TIOCREMOTE` on its read queue, it sends an `M_IOCACK` message downstream and the appropriate `M_CTL` message upstream to enable/disable canonical processing.
- When it receives an `M_DELAY` message on its read or write queue, it discards the message and does not act on it.
- When it receives an `M_IOCTL` message with type `JWINSIZE` on its write queue and if the values in the `jwinsize` structure of `ptem` are not zero, it sends an `M_IOCACK` message upstream with the `jwinsize` structure. If the values are zero, it sends an `M_IOCNAK` message upstream.
- When it receives an `M_IOCTL` message of type `TIOCGWINSZ` on its write queue and if the values in the `winsize` structure are not zero, it sends an `M_IOCACK` message upstream with the `winsize` structure. If the values are zero, it sends an `M_IOCNAK` message upstream. It also saves the information passed to it in the `winsize` structure and sends a `STREAMS` signal message for signal `SIGWINCH` upstream to the slave process if the size changed.
- When the `ptem` module receives an `M_IOCTL` message with type `TIOCGWINSZ` on its read queue and if the values in the `winsize` structure are not zero, it sends an `M_IOCACK` message downstream with the `winsize` structure. If the values are zero, it sends an `M_IOCNAK` message downstream. It also saves the information passed to it in the `winsize` structure and sends a `STREAMS` signal message for signal `SIGWINCH` upstream to the slave process if the size changed.
- All other messages not mentioned above are passed to the next module or driver.

Data Structure

SunOS 5.3 reserves the right to change `ptem`'s internal implementation. This structure should be relevant only to people wanting to change the module.

Each instantiation of the `pem` module is associated with a local area. These data is held in a structure called `pem` that has the following format:

```
struct pem
{
    long cflags;           /* copy of c_flags */
    mblk_t *dack_ptr;      /* pointer to preallocated msg blk
                           used to send disconnect */
    queue_t *q_ptr;        /* pointer to pem's read queue */
    struct winsize wsz; /*struct to hold windowing info*/
    unsigned short state; /* state of pem entry */
};
```

When the `pem` module is pushed onto the slave side Stream, a search of the `pem` structure is made for a free entry (*state* is not set to `INUSE`). The *c_flags* of the `termio` structure and the windowing variables are stored in *cflags* and *wsz* respectively. The *dack_ptr* is a pointer to a message block used to send a zero-length message whenever a hang-up occurs on the slave side.

Open and Close Routines

In the open routine of `pem` a STREAMS message block is allocated for a zero-length message for delivering a hang-up message; this allocation of a buffer is done before it is needed to ensure that a buffer is available. An `M_SETOPTS` message is sent upstream to set the read side Stream head queues, to assign high and low water marks (1024 and 256 respectively), and to establish a controlling terminal.

The same default values as for the line discipline module are assigned to *cflags*, and `INUSE` to the *state* field.

Note – These default values are currently being examined and may change in the future.

The open routine fails if:

- No free entries are found when the `pem` structure is searched.
- *sflag* is not set to `MODOPEN`.
- A zero-length message can not be allocated (no buffer is available).

- A `stroptions` structure can not be allocated.

The close routine is called on the last close of the slave side Stream. Pointers to read and write queue are cleared and the buffer for the zero-length message is freed.

Remote Mode

A feature known as *remote mode* is available with the pseudo-TTY subsystem. This feature is used for applications that perform the canonical function normally done by the `ldterm` module and TTY driver. The remote mode allows applications on the master side to turn off the canonical processing. An `ioctl TIOCREMOTE` with a nonzero parameter (`ioctl(fd, TIOCREMOTE, 1)`) is issued on the master side to enter the remote mode. When this occurs, an `M_CTL` message with the command `MC_NO_CANON` is sent to the `ldterm` module indicating that data should be passed when received on the read side and no canonical processing is to take place. The remote mode may be disabled by:

```
ioctl(fd, TIOCREMOTE, 0).
```

Packet Mode

The STREAMS-based pseudo-terminal subsystem also supports a feature called *packet mode*. This is used to inform the process on the master side when *state* changes have occurred in the pseudo-TTY. Packet mode is enabled by pushing the `pckt` module on the master side. Data written on the master side is processed normally. When data is written on the slave side or when other messages are encountered by the `pckt` module, a header is added to the message so it can be subsequently retrieved by the master side with a `getmsg` operation.

The `pckt` module does the following:

- When a message is passed to this module on its write queue, the module does no processing and passes the message to the next module or driver.
- The `pckt` module creates an `M_PROTO` message when one of the following messages is passed to it:

```
M_DATA, M_IOCTL, M_PROTO/M_PCPROTO, M_FLUSH,
M_START/M_STOP, M_STARTI/M_STOPI, and M_READ.
```

All other messages are passed through. The `M_PROTO` message is passed upstream and retrieved when the user issues `getmsg(2)`.

- If the message is an `M_FLUSH` message, `pckt` does the following:

If the flag is `FLUSHW`, it is changed to `FLUSHR` (because `FLUSHR` was the original flag before the `pts` driver changed it), packetized into an `M_PROTO` message, and passed upstream. To prevent the Stream head's read queue from being flushed, the original `M_FLUSH` message must not be passed upstream.

If the flag is `FLUSHR`, it is changed to `FLUSHW`, packetized into an `M_PROTO` message, and passed upstream. In order to flush of the write queues properly, an `M_FLUSH` message with the `FLUSHW` flag set is also sent upstream.

If the flag is `FLUSHRW`, the message with both flags set is packetized and passed upstream. An `M_FLUSH` message with the `FLUSHW` flag set is also sent upstream.

Pseudo-TTY Drivers - `ptm` and `pts`

To use the pseudo-TTY subsystem, a node for the master side driver `/dev/ptmx` and `N` number of slave drivers (`N` is determined at installation time) must be installed. The names of the slave devices are `/dev/pts/M` where `M` has the values 0 through `N-1`. A user accesses a pseudo-TTY device through the *master* device (called `ptm`) that in turn is accessed through the clone driver (see `clone(7)`). The *master* device is set up as a clone device where its major device number is the major for the clone device and its minor device number is the major for the `ptm` driver.

The *master* pseudo driver is opened via the `open(2)` system call with `/dev/ptmx` as the device to be opened. The clone open finds the next available minor device for that major device; a *master* device is available only if it and its corresponding *slave* device are not already open. There are no nodes in the file system for *master* devices.

When the *master* device is opened, the corresponding *slave* device is automatically locked out. No user may open that *slave* device until it is unlocked. A user may invoke a function `grantpt` that will change the owner of the *slave* device to that of the user who is running this process, change the group id to TTY, and change the mode of the device to `0620`. Once the

permissions have been changed, the device may be unlocked by the user. Only the owner or super-user can access the *slave* device. The user must then invoke the `unlockpt` function to unlock the *slave* device. Before opening the *slave* device, the user must call the `ptsname` function to obtain the name of the *slave* device. The functions `grantpt`, `unlockpt`, and `ptsname` are called with the file descriptor of the *master* device. The user may then invoke the `open` system call with the name that was returned by the `ptsname` function to open the *slave* device.

The following example shows how a user may invoke the pseudo-TTY subsystem:

```
int fdm fds;
char *slavename;
extern char *ptsname();

fdm = open("/dev/ptmx", O_RDWR);      /* open master */
grantpt(fdm);                        /* change permission of slave */
unlockpt(fdm);                       /* unlock slave */
slavename = ptsname(fdm);             /* get name of slave */
fds = open(slavename, O_RDWR); /* open slave */
ioctl(fds, I_PUSH, "ptem");          /* push ptem */
ioctl(fds, I_PUSH, "ldterm"); /* push ldterm */
```

Unrelated processes may open the pseudo device. The initial user may pass the master file descriptor using a STREAMS-based pipe or a slave name to another process to enable it to open the *slave*. After the *slave* device is open, the owner is free to change the permissions.

Note – Certain programs such as `write` and `wall` are set group-id (`setgid`) to TTY and are also able to access the slave device.

After both the *master* and *slave* have been opened, the user has two file descriptors that provide full-duplex communication using two Streams. The two Streams are automatically connected. The user may then push modules onto either side of the Stream. The user also needs to push the `ptem` and `ldterm` modules onto the slave side of the pseudo-terminal subsystem to get terminal semantics.

The *master* and *slave* drivers pass all STREAMS messages to their adjacent queues. Only the `M_FLUSH` needs some processing. Because the read queue of one side is connected to the write queue of the other, the `FLUSHR` flag is changed to `FLUSHW` flag and vice versa.

When the *master* device is closed, an `M_HANGUP` message is sent to the *slave* device which will render the device unusable. The process on the slave side gets the *errno* `ENXIO` when attempting to write on that Stream but it will be able to read any data remaining on the Stream head read queue. When all the data has been read, `read` returns 0 indicating that the Stream can no longer be used.

On the last close of the *slave* device, a zero-length message is sent to the *master* device. When the application on the master side issues a `read` or `getmsg` and 0 is returned, the user of the *master* device decides whether to issue a close that dismantles the pseudo-terminal subsystem. If the *master* device is not closed, the pseudo-TTY subsystem will be available to another user to open the *slave* device.

Since zero-length messages are used to indicate that the process on the slave side has closed and should be interpreted that way by the process on the master side, applications on the slave side should not write zero-length messages. If that occurs, the `write` returns 0, and the zero-length message is discarded by the `ptem` module.

The standard STREAMS system calls can access the pseudo-TTY devices. The *slave* devices support the `O_NDELAY` and `O_NONBLOCK` flags. Since the master side does not act like the terminal, if `O_NONBLOCK` or `O_NDELAY` is set, `read` on the master side returns -1 with *errno* set to `EAGAIN` if no data is available, and `write` returns -1 with *errno* set to `EAGAIN` if there is internal flow control.

The *master* driver supports the `ISPTM` and `UNLKPT` `ioctl`s that are used by the functions `grantpt`, `unlockpt`, and `ptsname` (see `grantpt(3C)`, `unlockpt(3C)`, `ptsname(3C)`). The `ioctl` `ISPTM` determines whether the file descriptor is that of an open *master* device. On success, it returns the major/minor number (type `dev_t`) of the *master* device which can be used to determine the name of the corresponding *slave* device. The `ioctl` `UNLKPT` unlocks the *master* and *slave* devices. It returns 0 on success. On failure, the *errno* is set to `EINVAL` indicating that the *master* device is not open.

The format of these commands is:

```
int ioctl (int fd, int command, int arg)
```

where `command` is either `ISPTM` or `UNLKPT` and `arg` is 0. On failure, -1 is returned.

When data is written to the master side, the entire block of data written is treated as a single line. The slave side process reading the terminal receives the entire block of data. Data is not input edited by the `ldterm` module regardless of the terminal mode. The master side application is responsible for detecting an interrupt character and sending an interrupt signal `SIGINT` to the process in the slave side. This can be done as follows:

```
ioctl (fd, TIOCSIGNAL, SIGINT)
```

where `SIGINT` is defined in the file `<signal.h>`. When a process on the master side issues this `ioctl`, the argument is the number of the signal that should be sent. The specified signal is then sent to the process group on the slave side.

To summarize, the *master* driver and *slave* driver have the following characteristics:

- Each *master* driver has one-to-one relationship with a *slave* device based on major/minor device numbers.
- Only one open is allowed on a *master* device. Multiple opens are allowed on the *slave* device according to standard file mode and ownership permissions.
- Each *slave* driver minor device has a node in the file system.
- An open on a *master* device automatically locks out an open on the corresponding *slave* driver.
- A *slave* cannot be opened unless the corresponding *master* is open and has unlocked the *slave*.
- To provide a TTY interface to the user, the `ldterm` and `pem` modules are pushed on the slave side.
- A close on the *master* sends a hang-up to the *slave* and renders both Streams unusable after all data have been consumed by the process on the slave side.
- The last close on the slave side sends a zero-length message to the *master* but does not sever the connection between the *master* and *slave* drivers.

grantpt

The `grantpt` function changes the mode and the ownership of the *slave* device that is associated with the given *master* device. Given a file descriptor *fd*, `grantpt` first checks that the file descriptor is that of the *master* device. If so, it obtains the name of the associated *slave* device and sets the user id to that of the user running the process and the group id to TTY. The mode of the *slave* device is set to *0620*.

If the process is already running as root, the permission of the *slave* can be changed directly without invoking this function. The interface is:

```
grantpt (int fd);
```

The `grantpt` function returns 0 on success and -1 on failure. It fails if one or more of the following occurs: *fd* is not an open file descriptor, *fd* is not associated with a *master* device, the corresponding *slave* could not be accessed, or a system call failed because no more processes could be created.

unlockpt

The `unlockpt` function clears a lock flag associated with a *master/slave* device pair. Its interface is:

```
unlockpt (int fd);
```

The `unlockpt` returns 0 on success and -1 on failure. It fails if one or more of the following occurs: *fd* is not an open file descriptor or *fd* is not associated with a *master* device.

ptsname

The `ptsname` function returns the name of the *slave* device that is associated with the given *master* device. It first checks that the file descriptor is that of the *master*. If it is, it then determines the name of the corresponding *slave* device `/dev/pts/M` and returns a pointer to a string containing the null-terminated path name. The return value points to static data whose content is overwritten by each call. The interface is:

```
char *ptsname (int fd);
```

The `ptsname` function returns a non-NULL path name upon success and a NULL pointer upon failure. It fails if one or more of the following occurs: `fd` is not an open file descriptor or `fd` is not associated with the *master* device.

MT STREAMS Overview

This chapter describes how to multi-thread a STREAMS driver or module. It covers the necessary conversion topics so that new and existing STREAMS modules and drivers will run in the multi-threaded kernel. We will be looking mostly at STREAMS specific multi-threading issues and techniques. Refer also to the *Writing Device Drivers* manual.

SunOS 5.x is a fully multi-threaded operating system able to make effective use of the available parallelism of a symmetric shared-memory multiprocessor computer. All kernel subsystems have been multi-threaded: scheduler, virtual memory, file systems, block/character/STREAMS I/O, networking protocols, and device drivers.

MT STREAMS requires you to use some new concepts and terminology. These concepts apply not only to STREAMS drivers, but to all device drivers in SunOS. For more a complete description of these terms, see *Writing Device Drivers*. Additionally, see Chapter 2, "Overview of STREAMS" of this guide for definitions, and Chapter 5, "Messages" for elements of MT drivers.

As an overview, you will need to understand the following terms and ideas.

Thread	sequence of instructions executed within context of a process
Lock	mechanism for restricting access to data structures
Single Threaded	restricting access to a single thread
Multi Threaded	allowing two or more threads access
Multiprocessing	two or more CPUs concurrently executing the OS

Concurrency	simultaneous execution
Preemption	suspending execution for the next thread to run
Monitor	portion of code that is single threaded
Mutual Exclusion	exclusive access to a data element by a single thread
Condition Variables	kernel event synchronization primitives
Counting Semaphores	memory based synchronization mechanism
Readers/Writer Locks	data lock allowing one writer and many readers
Callback	upon specific event, call module function

MT STREAMS Framework

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

A goal of SunOS 5.x is to preserve the interface and flavor of STREAMS to shield module code as much as possible from the impact of migrating to the multi-threaded kernel. The majority of the locking is hidden from the programmer and performed by the STREAMS kernel framework. As long as module code uses the standard, documented programmatic interfaces to shared kernel data structures (such as `queue_t`, `mblk_t`, and `dblk_t`), it will not have to explicitly lock these framework data structures.

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

The DDI/DKI entry points (open, close, put, and service procedures) plus certain callback procedures (scheduled with `qtimeout`, `qbufcall`, or `qwriter`) are termed *synchronous* entry points. All other entry points into a module are termed *asynchronous*. Examples of the latter are hardware interrupt routines, `timeout`, `bufcall`, and `esballoc` callback routines.

STREAMS Framework Integrity

The STREAMS framework guarantees the integrity of the STREAMS framework data structures, such as `queue_t`, `mbblk_t`, and `dbblk_t`, assuming the module conforms to the DDI/DKI thus does not directly access global operating system data structures nor facilities not described within the Driver-Kernel Interface.

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

As in previous SunOS releases, a module must not call another module's `put` or `service` procedures directly. The DDI/DKI routines `putnext()`, `put()`, and others in Section 9F must be used to pass the message to another queue. Calling another module's routines directly circumvents the design of the MT STREAMS framework and can yield unknown results.

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

Message Ordering

The STREAMS framework guarantees the ordering of messages along a stream if all the modules in the stream preserves message ordering internally. This ordering guarantee only applies to messages that are sent along the same stream and produced by the same source.

The STREAMS framework does not guarantee that a message has been seen by the next `put` procedure when `putnext()`, `qreply()` return.

Your MT Options

There are two MT configuration options available to a module (or driver):

- MT SAFE
- MT UNSAFE

MT SAFE modules

For MT SAFE mode it is possible to use MT STREAMS perimeters to restrict the concurrency in a module or driver to e.g.:

- Per module single-threading
- Per queue-pair single-threading
- Per queue single-threading
- Per queue or per queue-pair single-threading of the `put` and `service` procedures with per module single-threading of the `open` and `close` routines.
- Unrestricted concurrency in the `put` and `service` procedures with the ability to restrict the concurrency when handling messages that modify data.
- Completely unrestricted concurrency.

We recommend that you initially implement your module and configure it to be per-module single-threaded, and increase the level of concurrency as needed. The section "Sample Multi-threaded Device Driver" provides a complete example of using a per-module perimeter, and the section "Sample Multi-threaded Module with Outer perimeter" provides a complete example with a higher level of concurrency.

MT SAFE modules can use different MT STREAMS perimeters to restrict the concurrency in the module to a concurrency that is natural given the data structures that the module contains, thereby removing the need for module private locks. A module that requires unrestricted concurrency can be configured to have no perimeters. Such modules have to use explicit locking primitives to protect their data structures. While such modules can exploit the maximum level of concurrency allowed by the underlying hardware platform, they are more complex to develop and support. See the section on "MT SAFE Modules using Explicit Locks".

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

MT UNSAFE modules

MT UNSAFE mode for STREAMS modules is temporarily supported as an aid in porting SVR4 modules. MT UNSAFE might not be supported in future versions of the operating system. See the section on “MT UNSAFE Modules” on page 298 for details.

Preparing to Port

When modifying a STREAMS driver to take advantage of the multi-threaded kernel, a level of MT-safeness is selected according to:

- The desired degree of concurrency
- The natural concurrency of the underlying module
- The amount of effort/complexity required

Note that much of the effort in conversion is simply determining the appropriate degree of data sharing and the corresponding granularity of locking. The actual time spent configuring perimeters and/or installing locks should be much smaller than the time spent in analysis.

To port your module, you must understand the data structures used within your module as well as accesses to those data structures. It is your responsibility to fully understand the relationship between all portions of the module and private data within that module, and to use the MT STREAMS perimeters (or the synchronization primitives available) to maintain the integrity of these private data structures.

It is your responsibility to explicitly restrict access to private module data structures as appropriate to ensure the integrity of these data structures. You must use the MT STREAMS perimeters to restrict the concurrency in the module so that the parts of the module that modify module private data is single threaded with respect to the parts of the module that read the same data. Alternatively to the perimeters, you can use the synchronization primitives available (mutex, condition variables, readers/writer, semaphore) to explicitly restrict access to module private data appropriate for the operations within the module on that data.

The first step in multi-threading a module or driver is to analyze the module, breaking the entire module up into a list of individual operations and the private data structures referenced in each operation. Part of this first step is deciding upon a level of concurrency for the module. Ask yourself which of these operations can be multi threaded and which must be single threaded. Try to find a level of concurrency that is “natural” for the module and that matches one of the available perimeters (or alternatively, requires the minimal number of locks) and that has a simple and straightforward implementation. Avoid additional complexity. Avoid the desire to overly multi-thread the module at this point. Simple is better at this stage.

Typical questions to be answered are:

1. what data structures are maintained within the module?
2. what types of accesses are made to each field of these data structures?
3. when is each data structure accessed destructively (written) and when is it accessed non-destructively (read)?
4. which operations within the module should be allowed to execute concurrently?
5. is per-module single-threading appropriate for the module?
6. is per queue-pair or per queue single-threading appropriate?
7. what are the message ordering requirements?

Examples of natural levels of concurrency are:

- A module, where the `put` procedures read as well as modify module global data can be configured to be per module single-threaded using a per module inner perimeter.
- A module, where all the module private data is associated with a queue (or a read/write pair of queues) can be configured to be single-threaded for each queue (or queue pair) using the corresponding inner perimeter.
- A module where most of the module private data is associated with a queue (or a queue pair), but that in addition has some module global data which is mostly read, can be configured with a queue (or queue pair) inner perimeter plus an outer perimeter. The module can then use `qwriter()` to protect the sections where it modifies the module global data.

- A module that falls in one of the above categories, but requires higher concurrency for certain message types while not requiring message ordering, can be configured as one of the above perimeters with the addition of specifying shared inner perimeter access for the `put` procedures. The module can then use `qwriter()` when messages are handled in the `put` procedures that require exclusive access at the inner and/or outer perimeter.
- A hardware driver can use an appropriate set of inner and outer perimeters to restrict the concurrency in the `open`, `close`, `put`, and `service` procedures. Together with explicit synchronization primitives, these drivers restrict the concurrency when accessing the hardware registers in interrupt handlers etc. Such drivers need to be aware of the issues listed in the section "MT SAFE Modules using Explicit Locks".

Porting to SunOS 5.x

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

- SunOS 5.x Device Driver Interface (DDI/DKI).
- SunOS 5.x MT Design

For portability and correct operation, each module must adhere to the SunOS DDI/DKI. Several facilities available in previous releases of SunOS have changed and may take different arguments or provide different side-effects or may no longer exist in SunOS 5.x. The module writer should carefully review the module with respect to the DDI/DKI.

Each module that accesses underlying Sun-specific features included within SunOS should conform to the Device Driver Interface. The SunOS 5.x DDI defines the interface used by the device driver to register device hardware interrupts, access device node properties, map device slave memory, and establish and synchronize memory mappings for DVMA (Direct Virtual Memory Access). These areas are primarily applicable to hardware device drivers. Refer to the Device Driver Interface Specification within the *Writing Device Drivers* for details on the 5.x DDI and DVMA.

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

After reviewing the module for conformance to the SunOS 5.x DKI and DDI specifications, the module writer should be able to consider the impact of multi-threading on the module.

MT SAFE Modules

We recommend that your MT SAFE modules use perimeters and avoid using module private locks. Should you opt to use module private locks you need to read the section "MT SAFE Modules using Explicit Locks" in addition to this section.

MT STREAMS perimeters

Note – The support for MT STREAMS perimeters and related interfaces (`qwriter`, `qwait`, `qtimeout`, and `qbufcall`) is new to SunOS 5.3. These interfaces are subject to minor change based on further experience using these facilities.

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

Figure 13-1 and figure 13-2 are examples of inner perimeters, and figure 13-3 shows multiple inner perimeters inside an outer perimeter.

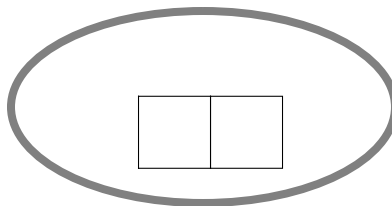


Figure 13-1 Inner perimeter spanning a pair of queues. (D_MPTQAIR)

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

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

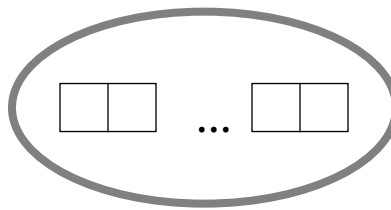


Figure 13-2 Inner perimeter spanning all queues in a module. (D_MTPERMOD)

When a thread is inside a perimeter and it calls `putnext()` (or `putnextctl()` etc.), it is possible that the thread will “loop around” through other STREAMS modules and try to re-enter a put procedure inside the original perimeter. If this re-entry conflicts with the earlier entry (e.g. if the first entry has exclusive access at the inner perimeter), the STREAMS framework will defer the re-entry while preserving the order of the messages attempting to enter the perimeter. Thus, `putnext()` will return without the message having been passed to the put procedure and the framework will pass in the message to the put procedure when it is possible to enter the perimeters.

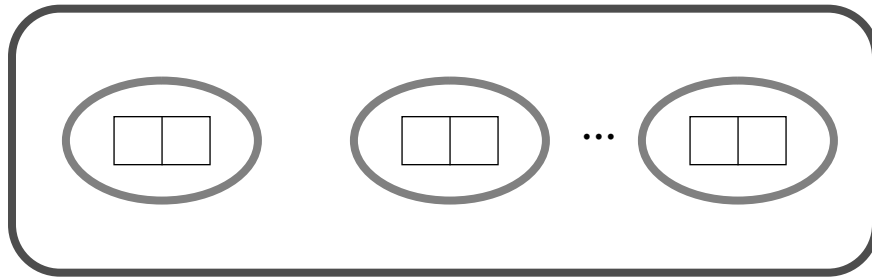


Figure 13-3 Outer perimeter spanning all queues in a module with inner perimeters spanning each pair of queues. (D_MTOUTPERIM combined with D_MTQPAIR)

The optional outer perimeter spans all queues in a module (see figure 13-3)

Perimeter options

There are several flags that are used to specify the perimeters. These flags fall into three categories:

- Define the presence and scope of the inner perimeter
- Define the presence of the outer perimeter (which can have only one scope)
- Modify the default concurrency for the different entry points

The inner perimeter is controlled by these mutually exclusive flags:

- D_MTPERMOD: The module has an inner perimeter that encloses all the module's queues.
- D_MTAPAIR: The module has an inner perimeter around each read/write pair of queues.
- D_MTPERQ: The module has an inner perimeter around each queue.
- None of the above: The module has no inner perimeter.

The presence of the outer perimeter is configured using:

- D_MTOUTPERIM: In addition to any inner perimeter (or none), the module has an outer perimeter that encloses all the module's queues. This can be combined with all the inner perimeter options except D_MTPERMOD.

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

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

MT configuration

To configure the driver as being MT SAFE, the `cb_ops` and `dev_ops` data structures must be initialized. This code must be in the header section of your module. For more information, see the example program in the section “Sample Multi-threaded Device Driver”, the code sample in Appendix E, “Configuration” and `cb_ops(9S)` and `dev_ops(9S)`.

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

qprocson()/qprocsoff()

The routines `qprocson()` and `qprocsoff()` respectively enable and disable the `put` and `service` procedures of the queue pair. Prior to the call to `qprocson`, and after the call to `qprocsoff`, the module’s `put` and `service` procedures are disabled; messages flow around the module as if it were not present in the Stream.

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

Note – To avoid deadlocks, modules should not hold private locks across the calls to `qprocson()` or `qprocsoff()`.

qtimeout()/qbufcall()

The `timeout()` and `bufcall()` callbacks are asynchronous, that is, they are not tracked by the STREAMS framework. For a module using MT STREAMS perimeters, this implies that the `timeout` and `bufcall` callback functions execute outside the scope of the perimeters. This makes it complex for the callbacks to synchronize with the rest of the module.

To make `timeout` and `bufcall` functionality easier to use for modules with perimeters, there are additional interfaces that use synchronous callbacks. These routines are `qtimeout(9F)`, `quntimeout(9F)`, `qbufcall(9F)`, and `qunbufcall(9F)`. When using these routines, the callback functions are executed inside the perimeters, i.e. with the same concurrency restrictions as the `put` and `service` procedures.

qwriter()

Modules can use the `qwriter(9F)` function to upgrade from shared to exclusive access at a perimeter. For example, a module with an outer perimeter can use `qwriter()` in the `put` or `service` procedures to upgrade to exclusive access at the outer perimeter. A module where the `put` procedures run with shared access at the inner perimeter (`D_MTPUTSHARED`) can use `qwriter()` in the `put` or `service` procedures to upgrade to exclusive access at the inner perimeter.

Note – Note that `qwriter()` cannot be used in the `open` or `close` procedures. If a module needs exclusive access at the outer perimeter in the `open` and/or `close` procedures, it has to specify that the outer perimeter should always be entered exclusively for `open` and `close` (using `D_MTOCEXCL`).

The STREAMS framework guarantees that all deferred `qwriter` callbacks associated with a queue have executed before the module's `close` routine is called for that queue.

For an example of a driver using `qwriter()` see the section “Sample Multi-threaded Module with Outer perimeter”.

qwait()

A module that uses perimeters and must wait in its `open` or `close` procedure for a message from another STREAMS module has to wait outside the perimeters; otherwise the message would never be allowed to enter its `put` and `service` procedures. This is accomplished by using the `qwait()` interface. See `qwriter(9F)` for an example.

Asynchronous Callbacks

Interrupt handlers and other asynchronous callback functions require special care by the module writer, since they can execute asynchronously to threads executing within the module `open`, `close`, `put`, and `service` procedures.

For modules using perimeters, we recommend using `qtimeout` and `qbufcall` instead of `timeout` and `bufcall`, since the `qtimeout` and `qbufcall` callbacks are synchronous and consequently introduce no special synchronization requirements.

Since a thread can enter the module at any time, the module writer is responsible for ensuring that the asynchronous callback function acquires the proper private locks before accessing private module data structures and then releases these locks before returning. It is the responsibility of the module writer to cancel any outstanding registered callback routines before the data structures on which the callback routines depend are deallocated and the module closed.

- For hardware device interrupts, this involves disabling the device interrupts.
- Outstanding callbacks from `timeout()` and `bufcall()` must be cancelled by calling `untimeout()` and `unbufcall()`.
- Outstanding callbacks from `esballoc()`, if associated with a particular Stream, must be allowed to complete before the module `close` routine deallocates those private data structures on which they depend.

Note – The module cannot hold certain private locks across calls to `untimeout()` or `unbufcall()`. These locks are those which the module's `timeout()` or `bufcall()` callback functions acquire. See section “MT SAFE Modules using Explicit Locks”.

Close Race Conditions

Since the callback functions are by nature asynchronous, they may be executing or about to execute at the time the module close routine is called. It is the responsibility of the module writer to cancel all outstanding callback and interrupt conditions before deallocating those data structures or returning from the close routine.

The callback functions scheduled with `timeout()` and `bufcall()` are guaranteed to have been cancelled by the time `untimeout()` and `unbufcall()` return. The same is true for `qtimeout()` and `qbufcall()` by the time `quntimeout()` and `qunbufcall()` return. You must also take responsibility for other asynchronous routines, including `esballoc()` callbacks and hardware as well as software interrupts.

Module unloading and esballoc

The STREAMS framework prevents a module/driver text from being unloaded while there are open instances of the module or driver. If a module does not cancel all callbacks in the last `close` routine it has to refuse to be unloaded.

This is an issue mainly for modules and drivers using `esballoc` since `esballoc` callbacks can not be cancelled. Thus modules and drivers using `esballoc` have to be prepared to handle calls to the `esballoc` callback free function after the last instance of the module or driver has been closed.

Modules and drivers can refuse to be unloaded by having their `_fini()` routine return `EBUSY`.

Use of `q_next`

The `q_next` field in the `queue_t` structure can be dereferenced in `open`, `close`, `put` and `service` procedures as well as the synchronous callback procedures (scheduled with `qtimeout()`, `qbufcall()`, and `qwriter()`).

All other module code, such as interrupt routines, `timeout()` and `esballoc()` callback routines, cannot dereference `q_next`. Those routines have to use the “next” version of all functions, that is, use e.g. `canputnext()` instead of dereferencing `q_next` and using `canput()`.

MT SAFE Modules using Explicit Locks

Although we recommend you use MT STREAMS perimeters you have the option of using explicit locks either instead of perimeters or in order to augment the concurrency restrictions provided by the perimeters.



Caution – Explicit locks can not, in general, be used to preserve message ordering in a module due to the risk of reentering the module. Use MT STREAMS perimeters to preserve message ordering.

All four types of kernel synchronization primitives are available to the module writer: mutexes, readers/writer locks, semaphores, and condition variables. Since `cv_wait()` implies a context switch, it can only be called from the module’s `open` and `close` procedures, which are executed with valid process context. It is the responsibility of the module writer to use the synchronization primitives provided to protect accesses and ensure the integrity of private module data structures.

Constraints when using locks

When adding locks in a module it is important to observe these constraints:

- Avoid holding module private locks across calls to `putnext()` etc., since the module might be reentered by the same thread that called `putnext()`, causing the module to try to acquire a lock that it already holds. This can cause kernel panic.

- Do not hold module private locks, acquired in `put` or `service` procedures, across the calls to `qprocson()` or `qprocoff()`. Doing this will cause deadlock, since `qprocson()` and `qprocoff()` wait until all threads leave the inner perimeter.
- Similarly, do not hold locks, acquired in the `timeout` and `bufcall` callback procedures, across the calls to `untimeout` or `unbufcall`. Doing this will cause deadlock, since `untimeout` and `unbufcall` wait until an already executing callback has completed.

The first restriction makes it very hard to use module private locks as a means of preserving message ordering. MT STREAMS perimeters is the preferred mechanism to preserve message ordering.

Preserving message ordering

Module private locks cannot be used to preserve message ordering, since they cannot be held across calls to `putnext()` (and the other messages that pass routines to other modules). The alternatives for preserving message ordering are:

- Use MT STREAMS perimeters.
- Pass all messages through the `service` procedures. The `service` procedure can drop the locks before calling `putnext()`, `qreply()` etc. without reordering messages, since the framework guarantees that at most one thread will execute in the `service` procedure for a given queue.

The use of perimeters is preferred since there is a performance penalty associated with using `service` procedures.

MT UNSAFE Modules

Most USL DDI/DKI compliant STREAMS drivers and modules can run without any source changes.

Note – This is not highly recommended nor 100% applicable, since this might jeopardize performance or possibly cause inoperability. These exceptions are usually due to specific implementation issues. It is expected that unsafe modules will run approximately as fast as they would have on a uniprocessor.

Note – SunOS supports an MT UNSAFE mode for STREAMS modules as an aid in porting modules. This feature should be considered a transition aid and may not be supported in future releases of the operating system. It is strongly recommended that all STREAMS modules and drivers are converted to be MT SAFE.

All MT UNSAFE code within the system runs single threaded, meaning there is no concurrency in the MT UNSAFE code. Only one executing thread is allowed within the MT UNSAFE code at any one time, with the exceptions described below. While the thread executing within the MT UNSAFE code can be preempted at any time, no other thread will be allowed entry into the MT UNSAFE code.

Modifying UNSAFE Drivers

By default, all STREAMS modules and drivers are considered MT UNSAFE unless configured into the system as MT SAFE (“D_MP”).

Unsafe drivers run with only the minimum of modification. They run under the general unsafe driver monitor, which implies that at any time, only one processor in the entire system is executing unsafe driver code. Thus, such modules do not gain any performance advantage by being run in a multiprocessor environment. Since these modules hold the mutex lock controlling entry to this monitor, they should not block for long periods, except by calling `sleep()`, which will transparently release and re-acquire the mutex for the caller.

Unsafe drivers are also the only kernel code that can call `sleep()` without catastrophic results. In general, such code will not explicitly block for any other reason other than `sleep()`, since the pre-MT kernel contained no locks.

Some module code cannot run safely as MT UNSAFE. Modules that access data shared by other modules must be converted, unless all other modules sharing such data is themselves unsafe. Also, modules that access safe modules by means other than `putnext()` and the like must be modified. This includes modules that call the `put` procedure of an other module directly.

Caveats

Preemption

The following events will allow the current thread to block, and another thread to enter the MT UNSAFE module, thus preempting the current thread:

- calling another module's `put` procedure via `putnext`, `putcctl`, `qreply`, ...
- `sleep()`
- `delay()`
- `strlog()`
- `cmn_err()`

Once entered, a thread within an MT UNSAFE module is allowed to execute until it returns or until it calls one of these routines. Other threads may have been allowed to execute within the module during the interim between calling one of the above routines and it returning. Consequently, the MT UNSAFE module must be prepared to save state across this preemptable point and revalidate private state information when the routine returns. This is not necessary if the module returns immediately after calling one of the above preemptable routines.

Asynchronous Callbacks

The MT STREAMS framework automatically restricts access into the MT UNSAFE module from all entry points. In addition to the MT UNSAFE module DKI entry points, the framework also blocks asynchronous callback routines entry into the module if a thread is currently active within the module until that thread exits the module. The following sources of asynchronous entry into the MT UNSAFE module are monitored by the framework and are not allowed to preempt a thread executing within the MT UNSAFE module:

- `timeout`
- `bufcall`
- `esballoc`
- software interrupt service routine
- `delay`

- device hardware interrupt service routine

Just like MT SAFE modules, MT UNSAFE modules have to cancel all outstanding callbacks in their `close` routine. See “Close Race Conditions” on page 296.

Interrupt Handlers

As described earlier, the framework singly threads all MT UNSAFE code within the system. The interrupt service routine is not called by the framework until any thread actively executing MT UNSAFE code within the system has exited the MT UNSAFE code. Therefore the MT UNSAFE module may not spin-wait for a hardware interrupt, since this interrupt handler is not called until the thread exits the module.

Sharing Data Structures

Modules that share some data structure(s) must be configured as either all MT UNSAFE or all MT SAFE. Mixing of module configurations is not allowed, since this would allow entry by multiple threads into the module.

New facilities

MT UNSAFE modules cannot use the regular synchronization primitives (such as mutexes and condition variables). Instead of condition-variables they have to use `sleep()` and `wakeup()`.

MT UNSAFE modules cannot use `put(9F)`.

Old Facilities

This section describes routines your unsafe module may call and how these translate into the new MT interfaces. Some translations are one-for-one, just using a new call in place of the older one. Others require new ways of viewing the problem and new techniques to solve them.

spl

Traditionally, modules have used the DKI `spl` routines to set the interrupt priority level of a processor to block certain hardware device interrupts. The intent of this was to block hardware interrupt preemption during a particular module operation so that the operation would effectively be atomic.

Prior to SunOS 5.x, only one active thread was allowed within the kernel at any one time. The only form of preemption in a pre-SunOS 5.x kernel came from device interrupts. Therefore using `spl` was a simple and effective method of single-threading in a pre-SunOS 5.x kernel.

In SunOS 5.x this is no longer true. The `spl` routines block only one form of preemption, those that arise directly from device hardware interrupts, and do not prevent preemption by other threads. Since the `spl` routines affect only one of the processors within the MP system, the device interrupt is not masked on other processors within the system. This can allow the hardware interrupt to be taken by any of the other processors in an MP system.

Use of the `spl` routines is restricted to MT UNSAFE modules only. The `spl` routines are not useful for MT UNSAFE drivers since the MT UNSAFE driver's interrupt handler will not be called as long as there is an active thread within the module.

MT SAFE modules should use the MT STREAMS perimeters or mutex, readers-writer, semaphore, and condition variable synchronization primitives instead of `spl` to prevent possible preemption of a non-atomic operation. MT SAFE modules must not call `spl`.

sleep/wakeup

In SunOS 5.x the functionality of `sleep/wakeup` is implemented via condition variables. The replacement for `sleep()` and `cv_signal()` is `cv_wait()`, while `cv_broadcast()` replace `wakeup()`. See *Writing Device Drivers* for details on using condition variables.

Since only the module `open` and `close` routines have user process context, the `cv_wait()` primitive can be called only from the module `open` and `close` routines. The `cv_signal()` and `cv_broadcast()` primitives can be called by the module at any time since they do not require valid user process context.

Modules that use MT STREAMS perimeters have to use `qwait()` instead of `cv_wait()` in order to allow their `put` or `service` procedures to be called while they are waiting.

Use of the routines `sleep()` and `wakeup()` are restricted to MT UNSAFE modules only, and should not be used by MT SAFE modules. MT SAFE modules should use condition variables or `qwait()` for this purpose.

Sample Multi-threaded Device Driver

Below is a sample multi-threaded, loadable, STREAMS pseudo-driver. The driver MT design is the simplest possible based on using a per module inner perimeter. Thus at most one thread can execute in the driver at any time. In addition, a `qtimeout()` synchronous callback routine is used. Note that the driver cancels any outstanding `qtimeout()` callback by using `quntimeout()` in the close routine. See “Close Race Conditions” on page 296.

Code Example 13-1 Sample Multi-threaded, Loadable, STREAMS Pseudo-Driver

```
/*
 * Example SunOS 5.x multi-threaded STREAMS pseudo device driver.
 * Using a D_MTPERMOD inner perimeter.
 */

#include      <sys/types.h>
#include      <sys/errno.h>
#include      <sys/stropts.h>
#include      <sys/stream.h>
#include      <sys/strlog.h>
#include      <sys/cmn_err.h>
#include      <sys/modctl.h>
#include      <sys/kmem.h>
#include      <sys/conf.h>
#include      <sys/ksynch.h>
#include      <sys/stat.h>
#include      <sys/ddi.h>
#include      <sys/sunddi.h>

/*
 * Function prototypes.
 */
static      int xxidentify(dev_info_t *);
static      int xxattach(dev_info_t *, ddi_attach_cmd_t);
```

```

static      int xxdetach(dev_info_t *, ddi_detach_cmd_t);
static      int xxgetinfo(dev_info_t *, ddi_info_cmd_t, void *, void**);
**);
static      int xxopen(queue_t *, dev_t *, int, int, cred_t *);
static      int xxclose(queue_t *, int, cred_t *);
static      int xxwput(queue_t *, mblk_t *);
static      int xxwsrv(queue_t *);
static      void xxtick(caddr_t);

/*
 * Streams Declarations
 */
static struct module_info xxm_info = {
    99,                /* mi_idnum */
    "xx",              /* mi_idname */
    0,                 /* mi_minpsz */
    INFPSZ,            /* mi_maxpsz */
    0,                 /* mi_hiwat */
    0,                 /* mi_lowat */
};

static struct qinit xxrinit = {
    NULL,              /* qi_putp */
    NULL,              /* qi_srvp */
    xxopen,            /* qi_qopen */
    xxclose,           /* qi_qclose */
    NULL,              /* qi_qadmin */
    &xxm_info,         /* qi_minfo */
    NULL               /* qi_mstat */
};

static struct qinit xxwinit = {
    xxwput,            /* qi_putp */
    xxwsrv,            /* qi_srvp */
    NULL,              /* qi_qopen */
    NULL,              /* qi_qclose */
    NULL,              /* qi_qadmin */
    &xxm_info,         /* qi_minfo */
    NULL               /* qi_mstat */
};

static struct streamtab xxstrtab = {
    &xxrinit,          /* st_rdinit */
    &xxwinit,          /* st_wrinit */
};

```

```

        NULL,                /* st_muxrinit */
        NULL                 /* st_muxwrinit */
    };

/*
 * define the xx_ops structure.
 */

static      struct cb_ops cb_xx_ops = {
    nodev,                /* cb_open */
    nodev,                /* cb_close */
    nodev,                /* cb_strategy */
    nodev,                /* cb_print */
    nodev,                /* cb_dump */
    nodev,                /* cb_read */
    nodev,                /* cb_write */
    nodev,                /* cb_ioctl */
    nodev,                /* cb_devmap */
    nodev,                /* cb_mmap */
    nodev,                /* cb_segmap */
    nochpoll,            /* cb_chpoll */
    ddi_prop_op,          /* cb_prop_op */
    &xxstrtab,            /* cb_stream */
    (D_NEW|D_MP|D_MTPERM) /* cb_flag */
};

static struct dev_ops xx_ops = {
    DEVO_REV,            /* devo_rev */
    0,                   /* devo_refcnt */
    xxgetinfo,            /* devo_getinfo */
    xxidentify,           /* devo_identify */
    nodev,               /* devo_probe */
    xxattach,            /* devo_attach */
    xxdetach,            /* devo_detach */
    nodev,               /* devo_reset */
    &cb_xx_ops,          /* devo_cb_ops */
    (struct bus_ops *)NULL /* devo_bus_ops */
};

/*
 * Module linkage information for the kernel.
 */
static struct modldrv modldrv = {

```

```

    &mod_driverops,    /* Type of module. This one is a driver */
    "xx",              /* Driver name */
    &xx_ops,            /* driver ops */
};

static struct modlinkage modlinkage = {
    MODREV_1,
    &modldrv,
    NULL
};

/*
 * Driver private data structure. One is allocated per Stream.
 */
struct xxstr {
    struct    xxstr *xx_next; /* pointer to next in list */
    queue_t  *xx_rq;         /* read side queue pointer */
    int       xx_minor;       /* minor device # (for clone) */
    int       xx_timeoutid;   /* id returned from timeout() */
};

/*
 * Linked list of opened Stream xxstr structures.
 * No need for locks protecting it since the whole module is
 * single threaded using the D_MTPERMOD perimeter.
 */
static struct xxstr    *xxup = NULL;

/*
 * Module Config entry points
 */

_init(void)
{
    return (mod_install(&modlinkage));
}

_fini(void)
{
    return (mod_remove(&modlinkage));
}

_info(struct modinfo *modinfop)

```



```

{
    return (mod_info(&modlinkage, modinfo));
}

/*
 * Auto Configuration entry points
 */

/*
 * Identify device.
 */
static int
xxidentify(dev_info_t *dip)
{
    if (strcmp(ddi_get_name(dip), "xx") == 0)
        return (DDI_IDENTIFIED);
    else
        return (DDI_NOT_IDENTIFIED);
}

/*
 * Attach device.
 */
static int
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    /*
     * This creates the device node.
     */
    if (ddi_create_minor_node(dip, "xx", S_IFCHR,
        ddi_get_instance(dip), DDI_PSEUDO, CLONE_DEV)
        == DDI_FAILURE) {
        return (DDI_FAILURE);
    }

    ddi_report_dev(dip);
    return (DDI_SUCCESS);
}

/*
 * Detach device.
 */
static int
xxdetach(dev_info_t *dip, ddi_detach_cmd_t cmd)

```

```

{
    ddi_remove_minor_node(dip, NULL);
    return (DDI_SUCCESS);
}

/* ARGSUSED */
static int
xxgetinfo(dev_info_t *dip, ddi_info_cmd_t infocmd, void *arg,
    void **resultp)
{
    dev_t dev = (dev_t) arg;
    int instance, ret;

    devstate_t *sp;
    state *statep;
    instance = getminor(dev);

    switch (infocmd) {
    case DDI_INFO_DEVT2DEVINFO:
        if ((sp = ddi_get_soft_state(statep,
            getminor((dev_t) arg))) != NULL) {
            *resultp = sp->devi;
            ret = DDI_SUCCESS;
        } else
            *resultp = NULL;
        break;

    case DDI_INFO_DEVT2INSTANCE:
        *resultp = (void *)instance;
        ret = DDI_SUCCESS;
        break;

    default:
        ret = DDI_FAILURE;
        break;
    }
    return (ret);
}

static
xxopen(rq, devp, flag, sflag, credp)
queue_t      *rq;
dev_t        *devp;
int           flag;

```

```

int          sflag;
cred_t       *credp;
{
    struct xxstr *xxp;
    struct xxstr **prevxxp;
    minor_t    minordev;

    /*
     * If this Stream already open - we're done.
     */
    if (rq->q_ptr)
        return (0);

    /*
     * Determine minor device number.
     */
    prevxxp = &xxup;
    if (sflag == CLONEOPEN) {
        minordev = 0;
        for (; (xxp = *prevxxp) != NULL;
              prevxxp = &xxp->xx_next) {
            if (minordev < xxp->xx_minor)
                break;
            minordev++;
        }
        *devp = makedevice(getmajor(*devp), minordev);
    } else
        minordev = getminor(*devp);

    /*
     * Allocate our private per-Stream data structure.
     */
    if ((xxp = kmem_alloc(sizeof (struct xxstr),
                          KM_SLEEP)) == NULL) {
        return (ENOMEM);
    }

    /*
     * Point q_ptr at it.
     */
    rq->q_ptr = WR(rq)->q_ptr = (char *) xxp;

    /*
     * Initialize it.

```

```

    */
    xxp->xx_minor = minordev;
    xxp->xx_timeoutid = 0;
    xxp->xx_rq = rq;

    /*
     * Link new entry into the list of active entries.
     */
    xxp->xx_next = *prevxxp;
    *prevxxp = xxp;

    /*
     * Enable xxput() and xxsrv() procedures on this queue.
     */
    qprocson(rq);

    return (0);
}

static
xxclose(rq, flag, credp)
queue_t      *rq;
int          flag;
cred_t       *credp;
{
    struct      xxstr      *xxp;
    struct      xxstr      **prevxxp;

    /*
     * Disable xxput() and xxsrv() procedures on this queue.
     */
    qprocsoff(rq);
    /*
     * Cancel any pending timeout.
     */
    xxp = (struct xxstr *) rq->q_ptr;
    if (xxp->xx_timeoutid != 0) {
        (void) qntimeout(rq, xxp->xx_timeoutid);
        xxp->xx_timeoutid = 0;
    }
    /*
     * Unlink per-Stream entry from the active list and free it.
     */

```

```

    for (prevxxp = &xxup; (xxp = *prevxxp) != NULL;
        prevxxp = &xxp->xx_next)
        if (xxp == (struct xxstr *) rq->q_ptr)
            break;
    *prevxxp = xxp->xx_next;
    kmem_free (xxp, sizeof (struct xxstr));

    rq->q_ptr = WR(rq)->q_ptr = NULL;

    return (0);
}

static
xxwput(wq, mp)
queue_t      *wq;
mblk_t       *mp;
{
    struct xxstr  *xxp = (struct xxstr *)wq->q_ptr;

    /* do stuff here */
    freemsg(mp);
    mp = NULL;

    if (mp != NULL)
        putnext(wq, mp);
}

static
xxwsrv(wq)
queue_t      *wq;
{
    mblk_t      *mp;
    struct xxstr  *xxp;

    xxp = (struct xxstr *) wq->q_ptr;

    while (mp = getq(wq)) {
        /* do stuff here */
        freemsg(mp);

        /* for example, start a timeout */
        if (xxp->xx_timeoutid != 0) {
            /* cancel running timeout */
            (void) quntimeout(wq, xxp->xx_timeoutid);

```

```

    }
    xxp->xx_timeoutid = qtimeout(wq, xxtick, (char *)xxp,
                                10);
}

static void
xxtick(arg)
    caddr_t arg;
{
    struct xxstr *xxp = (struct xxstr *)arg;

    xxp->xx_timeoutid = 0;      /* timeout has run */
    /* do stuff */
}

```

Sample Multi-threaded Module with Outer perimeter

Below is a sample multi-threaded, loadable, STREAMS module. The module MT design is a relatively simple one based on a per queue-pair inner perimeter plus an outer perimeter. The inner perimeter protects per-instance data structure (accessed through the `q_ptr` field) and the module global data is protected by the outer perimeter. The outer perimeter is configured so that the `open` and `close` routines have exclusive access to the outer perimeter. This is necessary since they both modify the global linked list of instances. Other routines that modify global data is run as `qwriter()` callbacks giving them exclusive access to the whole module.

```

/*
 * Example SunOS 5.x multi-threaded STREAMS module.
 * Using a per queue-pair inner perimeter plus an outer perimeter.
 */

#include <sys/types.h>
#include <sys/errno.h>
#include <sys/stropts.h>
#include <sys/stream.h>
#include <sys/strlog.h>
#include <sys/cmn_err.h>
#include <sys/kmem.h>
#include <sys/conf.h>

```

```

#include      <sys/ksynch.h>
#include      <sys/modctl.h>
#include      <sys/stat.h>
#include      <sys/ddi.h>
#include      <sys/sunddi.h>

/*
 * Function prototypes.
 */
static      int xxopen(queue_t *, dev_t *, int, int, cred_t *);
static      int xxclose(queue_t *, int, cred_t *);
static      int xxwput(queue_t *, mblk_t *);
static      int xxwsrv(queue_t *);
static      void xxwput_ioctl(queue_t *, mblk_t *);
static      int xxrput(queue_t *, mblk_t *);
static      void xxtick(caddr_t);

/*
 * Streams Declarations
 */
static struct module_info xxm_info = {
    99,                /* mi_idnum */
    "xx",              /* mi_idname */
    0,                 /* mi_minpsz */
    INFPSZ,           /* mi_maxpsz */
    0,                 /* mi_hiwat */
    0                  /* mi_lowat */
};

static struct qinit xxrinit = {
    xxrput,            /* qi_putp */
    NULL,              /* qi_srvp */
    xxopen,            /* qi_qopen */
    xxclose,           /* qi_qclose */
    NULL,              /* qi_qadmin */
    &xxm_info,         /* qi_minfo */
    NULL               /* qi_mstat */
};

static struct qinit xxwinit = {
    xxwput,            /* qi_putp */
    xxwsrv,            /* qi_srvp */
    NULL,              /* qi_qopen */
    NULL,              /* qi_qclose */

```

```

        NULL,                /* qi_qadmin */
        &xxm_info,           /* qi_minfo */
        NULL                 /* qi_mstat */
    };

    static struct streamtab xxstrtab = {
        &xxrinit,             /* st_rdinit */
        &xxwinit,            /* st_wrinit */
        NULL,                /* st_muxrinit */
        NULL                 /* st_muxwrinit */
    };

    /*
     * define the fmodsw structure.
     */

    static          struct fmodsw xx_fsw = {
        "xx",                /* f_name */
        &xxstrtab,           /* f_str */
        (D_NEW|D_MP|D_MTQPAIR|D_MTOUTPERIM|D_MTOCEXCL) /* f_flag */
    };

    /*

    /*
     * Module linkage information for the kernel.
     */
    static struct modlstrmod modlstrmod = {
        &mod_driverops,      /* Type of module; a STREAMS module */
        "xx module",         /* Module name */
        &xx_fsw,             /* fmodsw */
    };

    static struct modlinkage modlinkage = {
        MODREV_1,
        &modlstrmod,
        NULL
    };

    /*
     * Module private data structure. One is allocated per Stream.
     */
    struct xxstr {
        struct          xxstr *xx_next; /* pointer to next in list */
    };

```



```

    queue_t    *xx_rq;        /* read side queue pointer */
    int        xx_timeoutid; /* id returned from timeout() */
};

/*
 * Linked list of opened Stream xxstr structures and other module
 * global data. Protected by the outer perimeter.
 */
static struct xxstr    *xxup = NULL;
static int some_module_global_data;

/*
 * Module Config entry points
 */
int
_init(void)
{
    return (mod_install(&modlinkage));
}
int
_fini(void)
{
    return (mod_remove(&modlinkage));
}
int
_info(struct modinfo *modinfop)
{
    return (mod_info(&modlinkage, modinfop));
}

static int
xxopen(rq, devp, flag, sflag, credp)
queue_t    *rq;
dev_t      *devp;
int         flag;
int         sflag;
cred_t     *credp;
{
    struct xxstr *xvp;
    /*
     * If this Stream already open - we're done.
     */

```

```

if (rq->q_ptr)
    return (0);

if (sflag != MODOPEN)
    return (EINVAL);

/*
 * D_MTOCEXCL implies that the open and close routines have
 * exclusive access to the module global data structures.
 */

/*
 * Allocate our private per-Stream data structure.
 */
if ((xyp = kmem_alloc(sizeof (struct xxstr),
    KM_SLEEP)) == NULL) {
    return (ENOMEM);
}

/*
 * Point q_ptr at it.
 */
rq->q_ptr = WR(rq)->q_ptr = (char *) xyp;

/*
 * Initialize it.
 */
xyp->xx_rq = rq;
xyp->xx_timeoutid = 0;

/*
 * Link new entry into the list of active entries.
 */
xyp->xx_next = xxup;
xxup = xyp;

/*
 * Enable xyput() and xysrv() procedures on this queue.
 */
qprocson(rq);

return (0);
}

```

```

static int
xxclose(rq, flag, credp)
queue_t      *rq;
int           flag;
cred_t       *credp;

{
    struct      xxstr      *xxp;
    struct      xxstr      **prevxxp;

    /*
     * Disable xxput() and xxsrv() procedures on this queue.
     */
    qprocsoff(rq);
    /*
     * Cancel any pending timeout.
     */
    xxp = (struct xxstr *) rq->q_ptr;
    if (xxp->xx_timeoutid != 0) {
        (void) quntimeout(WR(rq), xxp->xx_timeoutid);
        xxp->xx_timeoutid = 0;
    }
    /*
     * D_MTOCEXCL implies that the open and close routines have
     * exclusive access to the module global data structures.
     */
    /*
     * Unlink per-Stream entry from the active list and free it.
     */
    for (prevxxp = &xxup; (xxp = *prevxxp) != NULL;
         prevxxp = &xxp->xx_next)
        if (xxp == (struct xxstr *) rq->q_ptr)
            break;
    *prevxxp = xxp->xx_next;
    kmem_free (xxp, sizeof (struct xxstr));

    rq->q_ptr = WR(rq)->q_ptr = NULL;
    return (0);
}

static int
xxrput(wq, mp)
queue_t      *wq;
mblk_t       *mp;

```

```

{
    struct xxstr    *xyp = (struct xxstr *)wq->q_ptr;

    /*
     * Do stuff here. Can read "some_module_global_data" since we
     * have shared access at the outer perimeter.
     */
    putnext(wq, mp);
}

/* qwriter callback function for handling M_IOCTL messages */
static void
xxwput_ioctl(wq, mp)
queue_t          *wq;
mblk_t           *mp;
{
    struct xxstr    *xyp = (struct xxstr *)wq->q_ptr;

    /*
     * Do stuff here. Can modify "some_module_global_data" since
     * we have exclusive access at the outer perimeter.
     */
    mp->b_datap->db_type = M_IOCNAK;
    greply(wq, mp);
}

static
xxwput(wq, mp)
queue_t          *wq;
mblk_t           *mp;
{
    struct xxstr    *xyp = (struct xxstr *)wq->q_ptr;

    if (mp->b_datap->db_type == M_IOCTL) {
        /* M_IOCTL will modify the module global data */
        qwriter(wq, mp, xxwput_ioctl, PERIM_OUTER);
        return;
    }
    /*
     * Do stuff here. Can read "some_module_global_data" since we
     * have shared access at the outer perimeter.
     */
    putnext(wq, mp);
}

```

```

static
xxwsrv(wq)
queue_t      *wq;
{
    mblk_t      *mp;
    struct xxstr *xsp;

    xsp = (struct xxstr *) wq->q_ptr;

    while (mp = getq(wq)) {
        /*
         * Do stuff here. Can read "some_module_global_data" since
         * we have shared access at the outer perimeter.
         */
        freemsg(mp);

        /* for example, start a timeout */
        if (xsp->xx_timeoutid != 0) {
            /* cancel running timeout */
            (void) qntimeout(wq, xsp->xx_timeoutid);
        }
        xsp->xx_timeoutid = qtimeout(wq, xxtick, (char *)xsp,
                                     10);
    }
}

static void
xxtick(arg)
caddr_t arg;
{
    struct xxstr *xsp = (struct xxstr *)arg;

    xsp->xx_timeoutid = 0;      /* timeout has run */
    /*
     * Do stuff here. Can read "some_module_global_data" since we
     * have shared access at the outer perimeter.
     */
}

```


STREAMS Data Structures



This appendix summarizes data structures commonly encountered in STREAMS module and driver development. Most of the data structures given in this appendix are contained in `<sys/stream.h>` and are documented in the man pages.

Many of the fields in the structures described below are intended for the private use of the STREAMS framework code. You should not permit modules and drivers to access these fields in any way, as their meaning, existence and size may change from release to release. *These fields may be omitted from the following descriptions.*

streamtab

This structure defines a module or a driver.

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

QUEUE Structures

Two sets of queue structures form a module. The structures are `queue`, `qinit`, `module_info`, and `module_stat` (optional).

queue

The `queue(9S)` structure has the following format:

```
struct qinit      *q_qinfo; /* procs and limits for queue */
struct msgb       *q_first; /* msg que head for this queue */
struct msgb       *q_last;  /* msg queue tail for this queue */
struct queue      *q_next;   /* next queue in Stream */
struct queue      *q_link    /* to next Q for scheduling */
void              *q_ptr;    /* to private data structure */
ulong             q_count;    /* number of bytes in queue */
ulong             q_flag;     /* queue state */
long              q_minpsz;   /* min packet size accepted */
long              q_maxpsz;   /* max packet size accepted */
ulong             q_hiwat;    /* queue high water mark */
ulong             q_lowat;    /* queue low water mark */
```

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

- `q_qinfo`: `st_rdinit` and `st_wrinit` (or `st_muxrinit` and `st_muxwinit`) - from `streamtab`
- `q_minpsz`, `q_maxpsz`, `q_hiwat`, `q_lowat` - from `module_info`
- `q_ptr` - optionally, by the driver/module open routine

Queue flags from `queue(9S)` for queue structure are defined as:

```
#define QENAB      0x001    /* Queue is already enabled to run */
#define QWANTR     0x002    /* Someone wants to read Q */
#define QWANTW     0x004    /* Someone wants to write Q */
#define QFULL      0x008    /* Q is considered full */
#define QREADR     0x010    /* This is the reader (first) Q */
#define QUSE       0x020    /* This queue in use (allocation) */
```



```

#define QNOENB          0x040      /* Don't enable Q via putq */
#define QOLD            0x080      /* Pre-SVR4 open/close interface */
#define QBACK          0x100      /* queue has been back-enabled */

```

qinit

`qinit(9S)` format is as follows:

```

struct qinit {
    int      (*qi_putp)();          /* put procedure */
    int      (*qi_srvp)();          /* service procedure */
    int      (*qi_qopen)();         /*called on each open or push*/
    int      (*qi_qclose)();        /*called on last close or pop*/
    int      (*qi_qadmin)();        /* reserved for future use */
    struct module_info *qi_minfo;    /* info struct */
    struct module_stat *qi_mstat;    /*stats struct (opt)*/
};

```

module_info

`module_info (9S)` has the following format:

```

struct module_info {
    ushort    mi_idnum;             /* module ID number */
    char      *mi_idname;           /* module name */
    long      mi_minpsz;            /* min packet size accepted */
    long      mi_maxpsz;            /* max packet size accepted */
    ulong     mi_hiwat;             /* high water mark, flow ctrl */
    ulong     mi_lowat;             /* low water mark, flow ctrl */
};

```

qband

The queue flow information, *qband* (9S) for each band is contained in the following structure:

```
/* Structure that describes the separate information
 * for each priority band in the queue
 */
struct qband {
    struct qband      *qb_next;          /* next band's info */
    ulong             qb_count;          /* number of bytes in band */
    struct msgb        *qb_first;        /* beginning of band's data */
    struct msgb        *qb_last;         /* end of band's data */
    ulong             qb_hiwat;          /* high water mark for band */
    ulong             qb_lowat;          /* low water mark for band */
    ulong             qb_flag;           /* QB_FULL, denotes that a
                                         band of data flow is flow
                                         controlled */
};
```

Message Structures

A message is composed of a linked list of triples, consisting of two structures (*msgb*(9S) and *datab*(9S)) and a data buffer.

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

    unsigned short     b_flag;            /*see below - Message flags*/
};typedef struct msgb mblk_t;
```

Note – Modules or drivers cannot modify `b_next` and `b_prev`. These fields are modified by utility routines such as `putq()` and `getq()`.

Conceptually the band belongs in the message block since it is associated with the message and not just with the data. However, the size of a message block is visible to modules and drivers, so the band is placed in the data block instead. Modules and drivers should have no knowledge of the size of the data block.

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

iocblk

This is contained in an `M_IOCTL` message block:

```
struct iocblk {
    int        ioc_cmd;           /* ioctl command type */
    cred_t     *ioc_cr;           /* full credentials */
    uint       ioc_id;            /* ioctl id */
    uint       ioc_count;         /* count of bytes in data field */
    int        ioc_error;         /* error code */
    int        ioc_rval;          /* return value */
};
```

copyreq

This is used in `M_COPYIN`/`M_COPYOUT` messages:

```
struct copyreq {
    int        cq_cmd;           /* ioctl command (from ioc_cmd) */
    cred_t     *cq_cr;           /* full credentials */
};
```

```

uint          cq_id;           /* ioctl id (from ioc_id) */
caddr_t       cq_addr;        /* address to copy data to/from */
uint          cq_size;        /* number of bytes to copy */
int           cq_flag;        /* see below */
mblk_t        *cq_private; /* private state information */
};

```

copyresp

This structure is used in M_IOCADATA:

```

struct copyresp {
    int          cp_cmd;        /* ioctl command (from ioc_cmd) */
    cred_t       *cp_cr;       /* full credentials */
    uint         cp_id;        /* ioctl id (from ioc_id) */
    caddr_t      cp_rval;      /* status of req; 0 for success
                               non-zero for failure */
    mblk_t       *cp_private; /* private state info */
};

```

Other Structures

strioc1

This structure supplies user values as an argument to the ioctl call I_STR in streamio(7).

```

struct strioc1 {
    int          ic_cmd;        /* downstream request */
    int          ic_timeout;    /* timeout acknowledgment-ACK/NAK */
    int          ic_len;        /* length of data argument */
    char         *ic_dp;        /* pointer to data argument */
};

```

linkblk

This structure is used in lower multiplexer drivers to indicate a link

```
struct linkblk {
    queue_t      *l_qtop;      /* lowest level write queue for upper */
                                /* Stream, set to NULL for persist links */
    queue_t      *l_qbot;      /* high level write q of lower Stream */
    int           l_index;      /* system-unique index for lower Stream */
};
```

stroptions

his structure holds various values used by the SREAMS. system:

```
struct stroptions {
    ulong         so_flags;      /* options to set */
    short         so_readopt;    /* read option */
    ushort        so_wroff;      /* write offset */
    long          so_minpsz;     /* minimum read packet size */
    long          so_maxpsz;     /* maximum read packet size */
    ulong         so_hiwat;      /* read queue high water mark */
    ulong         so_lowat;      /* read queue low water mark */
    unsigned char so_band;       /* band for water marks */
};
/* flags for Stream options set message */
#define SO_ALL          0x003f    /* set all options */
#define SO_READOPT      0x0001    /* set read option */
#define SO_WROFF        0x0002    /* set write offset */
#define SO_MINPSZ       0x0004    /* set minimum packet size */
#define SO_MAXPSZ       0x0008    /* set maximum packet size */
#define SO_HIWAT        0x0010    /* set high water mark */
#define SO_LOWAT        0x0020    /* set low water mark */
#define SO_MREADON      0x0040    /* set read notification on */
#define SO_MREADOFF     0x0080    /* set read notification off */
/*
#define SO_NDELOX       0x0100    /* old TTY semantics for NDELAY */
/*
#define SO_NDELOFF     0x0200    /* STREAMS semantics for NDELAY */
*/
```

```

#define SO_ISTTY          0x0400      /* Stream acting as
terminal*/
#define SO_ISNTTY        0x0800      /* Stream not acting as term*/
#define SO_TOSTOP        0x1000      /* stop on bkgrnd writes*/
#define SO_TONSTOP       0x2000      /* don't stop on bkgrnd jobs*/
#define SO_BAND          0x4000      /* water marks affect band */
#define SO_DELIM         0x8000      /* messages are delimited */
#define SO_NODELIM       0x010000    /* turn off delimiters */
#define SO_STRHOLD       0x020000    /* strwrite msg coalescing */

```

Introduction

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

STREAMS does not prevent a module or driver from generating any message type and sending it in any direction on the Stream. However, established processing and direction rules should be observed. Stream-head processing according to message type is fixed, although certain parameters can be altered.

The message types found in `<sys/stream.h>` are described in this appendix, classified according to their message queueing priority. Ordinary messages are described first, with high-priority messages following. In certain cases, two message types may perform similar functions, differing only in priority. Message construction is described in Chapter 5, “Messages”. The use of the word *module* will generally imply *module* or *driver*.

Ordinary messages are also called normal or non-priority messages. Ordinary messages are subject to flow control whereas high priority messages are not.

Ordinary Messages

M_BREAK

Sent to a driver to request that `BREAK` be transmitted on whatever media the driver is controlling.

The message format is not defined by STREAMS and its use is developer dependent. This message may be considered a special case of an `M_CTL` message. An `M_BREAK` message cannot be generated by a user-level process and is always discarded if passed to the Stream head.

M_CTL

Generated by modules that send information to a particular module or type of module. `M_CTL` messages are typically used for inter-module communication, as when adjacent STREAMS protocol modules negotiate the terms of their interface. An `M_CTL` message cannot be generated by a user-level process and is always discarded if passed to the Stream head.

M_DATA

Intended to contain ordinary data. Messages allocated by the `allocb()` routine (see Appendix C) 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(2)` and `putmsg(2)` system calls, the contents of `M_DATA` message blocks are referred to as the data part. Messages composed of multiple message blocks will typically have `M_DATA` as the message type for all message blocks following the first.

M_DELAY

Sent to a media driver to request a real-time delay on output. The data buffer associated with this message is expected to contain an integer to indicate the number of machine ticks of delay desired. `M_DELAY` messages are typically used to prevent transmitted data from exceeding the buffering capacity of slower terminals.

The message format is not defined by STREAMS and its use is developer dependent. Not all media drivers may understand this message. This message may be considered a special case of an `M_CTL` message. An `M_DELAY` message cannot be generated by a user-level process and is always discarded if passed to the Stream head.

M_IOCTL

Generated by the Stream head in response to `I_STR`, `I_LINK`, `I_UNLINK`, `I_PLINK`, and `I_PUNLINK` (`ioctl(2)` STREAMS system calls (see `streamio(7)`). Also generated in response to `ioctl` calls that contain a *command* argument value not defined in `streamio(7)`. When one of these `ioctls` is received from a user process, the Stream head uses values supplied in the call and values from the process to create an `M_IOCTL` message containing them, and sends the message downstream. `M_IOCTL` messages are intended to perform the general `ioctl` functions of character device drivers.

For an `I_STR` `ioctl`, 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_timeout;       /* 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_timeout* is the time the Stream head will wait for acknowledgment to the `M_IOCTL` message before timing out, and *ic_dp* is a pointer to an optional data buffer. On input, *ic_len* contains the length of the data in the buffer passed in and, on return from the call, it contains the length of the data, if any, being returned to the user in the same buffer.

The `M_IOCTL` message format is one `M_IOCTL` message block followed by zero or more `M_DATA` message blocks. STREAMS constructs an `M_IOCTL` message block by placing an `iocblk` structure, defined in `<sys/stream.h>`, in its data buffer. See Appendix A, “STREAMS Data Structures”, for a complete `iocblk` structure:

```
struct iocblk
{
    int ioc_cmd;           /* ioctl command type */
    ...
};
```

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

For an `I_STR` `ioctl`, *ioc_cmd* corresponds to *ic_cmd* of the `striocctl` structure. *ioc_cr* points to a credentials structure defining the user process's permissions (see `<cred.h>`). Its contents can be tested to determine if the user issuing the `ioctl` call is authorized to do so. For an `I_STR` `ioctl`, *ioc_count* is the number of data bytes, if any, contained in the message and corresponds to *ic_len*.

ioc_id is an identifier generated internally, and is used by the Stream head to match each `M_IOCTL` message sent downstream with response messages sent upstream to the Stream head. The response message that completes the Stream head processing for the `ioctl` is an `M_IOCACK` (positive acknowledgment) or an `M_IOCNAK` (negative acknowledgment) message.

For an `I_STR` `ioctl`, if a user supplies data to be sent downstream, the Stream head copies the data, pointed to by *ic_dp* in the `striocctl` structure, into `M_DATA` message blocks and links the blocks to the initial `M_IOCTL` message block. *ioc_count* is copied from *ic_len*. If there are no data, *ioc_count* is zero.

If the Stream head does not recognize the *command* argument of an `ioctl`, the head creates a transparent `M_IOCTL` message. The format of a transparent `M_IOCTL` message is one `M_IOCTL` message block followed by one `M_DATA` block. The form of the `iocblk` structure is the same as above. However, *ioc_cmd* is set to the value of the *command* argument in the `ioctl` system call and *ioc_count* is set to `TRANSPARENT`, defined in `<sys/stream.h>`. `TRANSPARENT` distinguishes the case where an `I_STR` `ioctl` may specify a value of *ioc_cmd* equivalent to the *command* argument of a transparent `ioctl`. The `M_DATA` block of the message contains the value of the *arg* parameter in the `ioctl` call.

The first module or driver that understands the *ioc_cmd* request contained in the `M_IOCTL` acts on it. For an `I_STR` `ioctl`, this action generally includes an immediate upstream transmission of an `M_IOCACK` message. For transparent `M_IOCTL`s, this action generally includes the upstream transmission of an `M_COPYIN` or `M_COPYOUT` message.

Intermediate modules that do not recognize a particular request must pass the message on. If a driver does not recognize the request, or the receiving module can not acknowledge it, an `M_IOCNAK` message must be returned.

`M_IOCACK` and `M_IOCNAK` message types have the same format as an `M_IOCTL` message and contain an `iocblk` structure in the first block. An `M_IOCACK` block may be linked to following `M_DATA` blocks. If one of these messages reaches the Stream head with an identifier that does not match that of the currently-outstanding `M_IOCTL` message, the response message is discarded. A common means of assuring that the correct identifier is returned is for the replying module to convert the `M_IOCTL` message into the appropriate response type and set `ioc_count` to 0, if no data is returned. Then, the `qreply()` utility (see Appendix C) is used to send the response to the Stream head.

In an `M_IOCACK` or `M_IOCNAK` message, `ioc_error` holds any return error condition set by a downstream module. If this value is non-zero, it is returned to the user in `errno`. Note that both an `M_IOCNAK` and an `M_IOCACK` may return an error. However, only an `M_IOCACK` can have a return value. For an `M_IOCACK`, `ioc_rval` holds any return value set by a responding module. For an `M_IOCNAK`, `ioc_rval` is ignored by the Stream head.

If a module processing an `I_STR ioctl` is sending data to a user process, it must use the `M_IOCACK` message that it constructs such that the `M_IOCACK` block is linked to one or more following `M_DATA` blocks containing the user data. The module must set `ioc_count` to the number of data bytes sent. The Stream head places the data in the address pointed to by `ic_dp` in the user `I_STR striocblk` structure.

If a module processing a transparent `ioctl` that is, it received a transparent `M_IOCTL`) wants to send data to a user process, it can use only an `M_COPYOUT` message. For a transparent `ioctl`, no data can be sent to the user process in an `M_IOCACK` message. All data must have been sent in a preceding `M_COPYOUT` message. The Stream head will ignore any data contained in an `M_IOCACK` message (in `M_DATA` blocks) and will free the blocks.

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

The Stream head blocks the user process until an `M_IOCACK` or `M_IOCNAK` response to the `M_IOCTL` (same `ioc_id`) is received. For an `M_IOCTL` generated from an `I_STR ioctl`, the Stream head will *time out* if no response is received

in *ic_timeout* interval (the user may specify an explicit interval or specify use of the default interval). For `M_IOCTL` messages generated from all other `ioctl`s, the default (infinite) is used.

M_PASSFP

Used by STREAMS to pass a file pointer from the Stream head at one end of a Stream pipe to the Stream head at the other end of the same Stream pipe.

The message is generated as a result of an `I_SENDFD` `ioctl` (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 that is, it is not seen by any modules or drivers in the Stream). This message should never be present on any queue except the read queue of a Stream head. Consequently, modules and drivers do not need to recognize this message, and it can be ignored by module and driver developers.

M_PROTO

Intended to contain control information and associated data. The message format is one or more (see note) `M_PROTO` message blocks followed by zero or more `M_DATA` message blocks as shown in Figure B-1. 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(2)` and `putmsg(2)` system calls, the control and data parts are passed separately.

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

Although its use is not recommended, the format of `M_PROTO` and `M_PCPROTO` (generically `PROTO`) messages sent upstream to the Stream head allows multiple `PROTO` blocks at the beginning of the message. `getmsg(2)` will compact the blocks into a single control part when passing them to the user process.

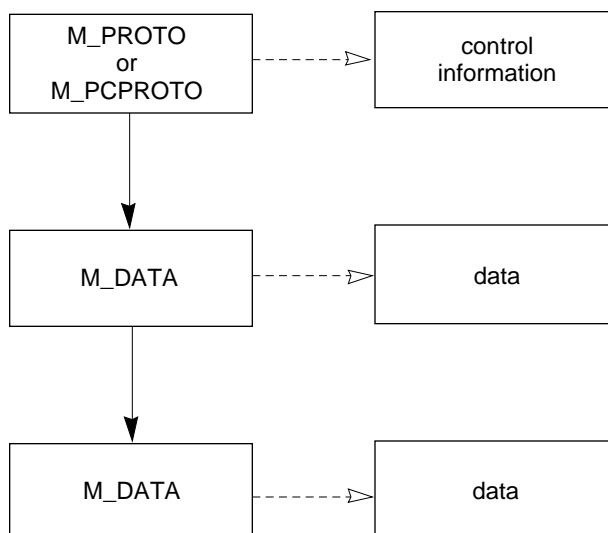


Figure B-1 `M_PROTO` and `M_PCPROTO` Message Structure

`M_RSE`

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

M_SETOPTS

Used to alter some characteristics of the Stream head. It is generated by any downstream module, and is interpreted by the Stream head. The data buffer of the message has the following structure as defined in `stream.h`. See Appendix A, “STREAMS Data Structures” for a complete `stroptions` structure.

[illegible]

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 `stropoptions` structure.
- **SO_READOPT:** Set the read mode (see `read(2)`) as specified by the value of *so_readopt* to:
 - **RNORM** byte stream
 - **RMSGD** message discard
 - **RMSGN** message non-discard
 - **RPROTNORM** normal protocol),
 - **RPROTDAT** turn `M_PROTO` and `M_PCPROTO` msgs into `M_DATA` msgs
 - **RPROTDIS** discard `M_PROTO` and `M_PCPROTO` blocks in a msg
 - and retain any linked `M_DATA` blocks
- **SO_WROFF:** Direct the Stream head to insert an offset (unwritten area, see *Write Offset* in Chapter 5, “Messages”) specified by *so_wroff* into the first message block of all `M_DATA` messages created as a result of a `write(2)`

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 `msgb` structure is greater than `db_base` in the `datab` 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` structure, Appendix A, “STREAMS Data Structures”). This value is advisory for the module immediately below the Stream head. It is intended to limit the size of `M_DATA` messages that the module should put to the Stream head. There is no intended minimum size for other message types. The default value in the Stream head is zero.
- `SO_MAXPSZ`: Change the maximum packet size value associated with the Stream head read queue to `so_maxpsz` (see `q_maxpsz` in the `queue` structure, Appendix A, “STREAMS Data Structures”). This value is advisory for the module immediately below the Stream head. It is intended to limit the size of `M_DATA` messages that the module should put to the Stream head. There is no intended maximum size for other message types. The default value in the Stream head is `INFPSZ`, the maximum STREAMS allows.
- `SO_HIWAT`: Change the flow control high water mark (`q_hiwat` in the `queue` structure, `qb_hiwat` in the `qband` structure) on the Stream head read queue to the value specified in `so_hiwat`.
- `SO_LOWAT`: Change the flow control low water mark (`q_lowat` in the `queue` structure, `qb_lowat` in the `qband` structure) on the Stream head read queue to the value specified in `so_lowat`.
- `SO_MREADON`: Enable the Stream head to generate `M_READ` messages when processing a `read(2)` system call. If both `SO_MREADON` and `SO_MREADOFF` are set in `so_flags`, `SO_MREADOFF` will have precedence.
- `SO_MREADOFF`: Disable the Stream head generation of `M_READ` messages when processing a `read(2)` system call. This is the default. If both `SO_MREADON` and `SO_MREADOFF` are set in `so_flags`, `SO_MREADOFF` will have precedence.

- **SO_NDELAY:** Set non-STREAMS TTY semantics for `O_NDELAY` (or `O_NONBLOCK`) processing on `read(2)` and `write(2)` system calls. If `O_NDELAY` (or `O_NONBLOCK`) is set, a `read(2)` will return 0 if no data is waiting to be read at the Stream head. If `O_NDELAY` (or `O_NONBLOCK`) is clear, a `read(2)` will block until data become available at the Stream head. (See note below)

Regardless of the state of `O_NDELAY` (or `O_NONBLOCK`), a `write(2)` will block on flow control and will block if buffers are not available.

If both `SO_NDELAY` and `SO_NDELOFF` are set in *so_flags*, `SO_NDELOFF` will have precedence.

Note – For conformance with the POSIX standard, it is recommended that new applications use the `O_NONBLOCK` flag whose behavior is the same as that of `O_NDELAY` unless otherwise noted.

- **SO_NDELOFF:** Set STREAMS semantics for `O_NDELAY` (or `O_NONBLOCK`) processing on `read(2)` and `write(2)` system calls. If `O_NDELAY` (or `O_NONBLOCK`) is set, a `read(2)` will return -1 and set `EAGAIN` if no data is waiting to be read at the Stream head. If `O_NDELAY` (or `O_NONBLOCK`) is clear, a `read(2)` will block until data become available at the Stream head. (See note above)

If `O_NDELAY` (or `O_NONBLOCK`) is set, a `write(2)` will return -1 and set `EAGAIN` if flow control is in effect when the call is received. It will block if buffers are not available. If `O_NDELAY` (or `O_NONBLOCK`) is set and part of the buffer has been written and a flow control or buffers not available condition is encountered, `write(2)` will terminate and return the number of bytes written.

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

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

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

- **SO_BAND:** Set watermarks in a band. If the **SO_BAND** flag is set with the **SO_HIWAT** or **SO_LOWAT** flag, the *so_band* field contains the priority band number the *so_hiwat* and *so_lowat* fields pertain to.

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

Only one band's watermarks can be updated with a single **M_SETOPTS** message.

- **SO_ISTTY:** Inform the Stream head that the Stream is acting like a controlling terminal.
- **SO_ISNTTY:** Inform the Stream head that the Stream is no longer acting like a controlling terminal.

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

- **SO_TOSTOP:** Stop on background writes to the Stream.
- **SO_TONSTOP:** Do not stop on background writes to the Stream. **SO_TOSTOP** and **SO_TONSTOP** are used in conjunction with job control.
- **SO_DELIM:** Messages are delimited.
- **SO_NODELIM:** Messages are not delimited.
- **SO_STRHOLD:** Enable strwrite message coalescing.

M_SIG

Sent upstream by modules or drivers to post a signal to a process. When the message reaches the front of the Stream head read queue, it evaluates the first data byte of the message as a signal number, defined in `<sys/signal.h>`. (The signal is not generated until it reaches the front of the Stream head read queue.) The associated signal will be sent to process(es) under the following conditions:

- If the signal is **SIGPOLL**, it will be sent only to those processes that have explicitly registered to receive the signal (see **I_SETSIG** in `streamio(7)`).

- If the signal is not `SIGPOLL` and the Stream containing the sending module or driver is a controlling TTY, the signal is sent to the associated process group. A Stream becomes the controlling TTY for its process group if, on `open(2)`, a module or driver sends an `M_SETOPTS` message to the Stream head with the `SO_ISTTY` flag set.
- If the signal is not `SIGPOLL` and the Stream is not a controlling TTY, no signal is sent, except in case of `SIOCSPGRP` and `TIOCSPGRP`. These two `ioctl`s set the process group field in the Stream head so the Stream can generate signals even if it is not a controlling TTY.

High-Priority Messages

M_COPYIN

Generated by a module or driver and sent upstream to request that the Stream head perform a `copyin()` on behalf of the module or driver. It is valid only after receiving an `M_IOCTL` message and before an `M_IOCACK` or `M_IOCNAK`.

The message format is one `M_COPYIN` message block containing a `copyreq` structure, defined in `<sys/stream.h>`. See Appendix A, “STREAMS Data Structures” for a complete `copyreq` structure.

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

The first four members of the structure correspond to those of the `iocblk` structure in the `M_IOCTL` message that allows the same message block to be reused for both structures. The Stream head will guarantee that the message block allocated for the `M_IOCTL` message is large enough to contain a `copyreq` structure. The `cq_addr` field contains the user space address from

which the data is to be copied. The *cq_size* field is the number of bytes to copy from user space. The *cq_flag* field is reserved for future use and should be set to zero.

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

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

M_COPYOUT

Generated by a module or driver and sent upstream to request that the Stream head perform a *copyout()* on behalf of the module or driver. It is valid only after receiving an *M_IOCTL* message and before an *M_IOCACK* or *M_IOCNAK*.

The message format is one *M_COPYOUT* message block followed by one or more *M_DATA* blocks. The *M_COPYOUT* message block contains a *copyreq* structure as described in the *M_COPYIN* message with the following differences: The *cq_addr* field contains the user space address to which the data is to be copied. The *cq_size* field is the number of bytes to copy to user space.

Data to be copied to user space is contained in the linked *M_DATA* blocks.

This message should not be queued by a module or driver unless it processes the data for the *ioctl* in some way.

M_ERROR

Sent upstream by modules or drivers to report some downstream error condition. When the message reaches the Stream head, the Stream is marked so that all subsequent system calls issued to the Stream, excluding *close(2)* and *poll(2)*, 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 *FLUSHRW* is sent downstream.

The Stream head maintains two error fields, one for the read-side and one for the write-side. The one-byte format `M_ERROR` message sets both of these fields to the error specified by the first byte in the message.

The second style of the `M_ERROR` message is two bytes long. The first byte is the read error and the second byte is the write error. This allows modules to set a different error on the read-side and write-side. If one of the bytes is set to `NOERROR`, then the field for the corresponding side of the Stream is unchanged. This allows a module to just an error on one side of the Stream. For example, if the Stream head was not in an error state and a module sent an `M_ERROR` message upstream with the first byte set to `EPROTO` and the second byte set to `NOERROR`, all subsequent read-like system calls (for example, `read`, `getmsg`) will fail with `EPROTO`, but all write-like system calls (for example, `write`, `putmsg`) will still succeed. If a byte is set to 0, the error state is cleared for the corresponding side of the Stream. The values `NOERROR` and 0 are not valid for the one-byte form of the `M_ERROR` message.

M_FLUSH

Requests all modules and drivers that receive it to flush their message queues (discard all messages in those queues) as indicated in the message. An `M_FLUSH` can originate at the Stream head, or in any module or driver. The first byte of the message contains flags that specify one of the following actions:

- `FLUSHR`: Flush the read queue of the module.
- `FLUSHW`: Flush the write queue of the module.
- `FLUSHRW`: Flush both the read queue and the write queue of the module.
- `FLUSHBAND`: Flush the message according to the priority associated with the band.

Each module passes this message to its neighbor after flushing its appropriate queue(s), until the message reaches one of the ends of the Stream.

Drivers are expected to include the following processing for `M_FLUSH` messages. When an `M_FLUSH` message is sent downstream through the write queues in a Stream, the driver at the Stream end discards it if the message action indicates that the read queues in the Stream are not to be flushed (only

FLUSHW set). If the message indicates that the read queues are to be flushed, the driver shuts off the FLUSHW flag, and sends the message up the Stream's read queues.

When a flush message is sent up a Stream's read-side, the Stream head checks to see if the write-side of the Stream is to be flushed. If only FLUSHR is set, the Stream head discards the message. However, if the write-side of the Stream is to be flushed, the Stream head sets the M_FLUSH flag to FLUSHW and sends the message down the Stream's write side. *All modules that queue messages must identify and process this message type.*

If FLUSHBAND is set, the second byte of the message contains the value of the priority band to flush.

M_HANGUP

Sent upstream by a driver to report that it can no longer send data upstream. As example, this might be due to an error, or to a remote line connection being dropped. When the message reaches the Stream head, the Stream is marked so that all subsequent `write(2)` and `putmsg(2)` system calls issued to the Stream will fail and return an `ENXIO` error. Those `ioctl`s 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(2)` or `getmsg(2)` calls to the Stream will not generate an error. These calls will return any messages (according to their function) that were on, or in transit to, the Stream head read queue before the `M_HANGUP` message was received. When all such messages have been read, `read(2)` will return 0 and `getmsg(2)` will set each of its two length fields to 0.

This message also causes a `SIGHUP` signal to be sent to the controlling process instead of the foreground process group, since the allocation and deallocation of controlling terminals to a session is the responsibility of the controlling process.

M_IOCACK

Signals the positive acknowledgment of a previous *M_IOCTL* message. The message format is one *M_IOCACK* block (containing an *iocblk* structure, see *M_IOCTL*) followed by zero or more *M_DATA* blocks. The *iocblk* data structure may contain a value in *ioc_rval* to be returned to the user process. It may also contain a value in *ioc_error* to be returned to the user process in *errno*.

If this message is responding to an *I_STR* *ioctl* (see *streamio(7)*), it may contain data from the receiving module or driver to be sent to the user process. In this case, message format is one *M_IOCACK* block followed by one or more *M_DATA* blocks containing the user data. The Stream head returns the data to the user if there is a corresponding outstanding *M_IOCTL* request. Otherwise, the *M_IOCACK* message is ignored and all blocks in the message are freed.

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

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

M_IOCDATA

Generated by the Stream head and sent downstream as a response to an *M_COPYIN* or *M_COPYOUT* message. The message format is one *M_IOCDATA* message block followed by zero or more *M_DATA* blocks. The *M_IOCDATA* message block contains a *copyresp* structure, defined in *<sys/stream.h>*. See Appendix A, "STREAMS Data Structures" for a complete *copyresp* structure.

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

The first three members of the structure correspond to those of the `iocblk` structure in the `M_IOCTL` message that allows the same message blocks to be reused for all of the related transparent messages (`M_COPYIN`, `M_COPYOUT`, `M_IOCACK`, `M_IOCNAK`). The `cp_rval` field contains the result of the request at the Stream head. Zero indicates success and non-zero indicates failure. If failure is indicated, the module should not generate an `M_IOCNAK` message. It must abort all `ioctl` processing, clean up its data structures, and return.

The `cp_private` field is copied from the `cq_private` field in the associated `M_COPYIN` or `M_COPYOUT` message. It is included in the `M_IOCDATA` message so the message can be self-describing. This is intended to simplify `ioctl` processing by modules and drivers.

If the message is in response to an `M_COPYIN` message and success is indicated, the `M_IOCDATA` block will be followed by `M_DATA` blocks containing the data copied in.

If an `M_IOCDATA` block is reused, any unused fields defined for the resultant message block should be cleared (particularly in an `M_IOCACK` or `M_IOCNAK`).

This message should not be queued by a module or driver unless it processes the data for the `ioctl` in some way.

M_IOCNAK

Signals the negative acknowledgment (failure) of a previous `M_IOCTL` message. Its form is one `M_IOCNAK` block containing an `iocblk` data structure (see `M_IOCTL`). The `iocblk` structure may contain a value in `ioc_error` to be returned to the user process in `errno`. Unlike the `M_IOCACK`, no user data or return value can be sent with this message. If any `M_DATA` blocks follow the `M_IOCNAK` block, the Stream head will ignore and free them. When the Stream head receives an `M_IOCNAK`, the outstanding `ioctl` request, if any, will fail. The format and usage of this message type is described further under `M_IOCTL`.

M_PCPROTO

As the `M_PROTO` message type, except for the priority and the following additional attributes.

When an `M_PCPROTO` message is placed on a queue, its `service` procedure is always enabled. The Stream head will allow only one `M_PCPROTO` message to be placed in its read queue at a time. If an `M_PCPROTO` message is already in the queue when another arrives, the second message is silently discarded and its message blocks freed.

This message is intended to allow data and control information to be sent outside the normal flow control constraints.

The `getmsg(2)` and `putmsg(2)` system calls refer to `M_PCPROTO` messages as high priority messages.

M_PCRSE

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

M_PCSIG

As the `M_SIG` message, except for the priority.

`M_PCSIG` is often preferable to the `M_SIG` message especially in TTY applications, because `M_SIG` may be queued while `M_PCSIG` is more guaranteed to get through quickly. For example, if one generates an `M_SIG` message when the `DEL` (delete) key is pressed on the terminal and one has already typed ahead, the `M_SIG` message becomes queued and the user doesn't get the call until it's too late; it becomes impossible to kill or interrupt a process by pressing a delete key.

M_READ

Generated by the Stream head and sent downstream for a `read(2)` system call if no messages are waiting to be read at the Stream head and if read notification has been enabled. Read notification is enabled with the `SO_MREADON` flag of the `M_SETOPTS` message and disabled by use of the `SO_MREADOFF` flag.

The message content is set to the value of the *nbyte* parameter (the number of bytes to be read) in the `read(2)` call.

`M_READ` is intended to notify modules and drivers of the occurrence of a `read`. It is also intended to support communication between Streams that reside in separate processors. The use of the `M_READ` message is developer dependent. Modules may take specific action and pass on or free the `M_READ` message. Modules that do not recognize this message must pass it on. All other drivers may or may not take action and then free the message.

This message cannot be generated by a user-level process and should not be generated by a module or driver. It is always discarded if passed to the Stream head.

M_START and M_STOP

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

The message format is not defined by STREAMS and its use is developer dependent. These messages may be considered special cases of an `M_CTL` message. These messages cannot be generated by a user-level process and each is always discarded if passed to the Stream head.

M_STARTI and M_STOPI

As `M_START` and `M_STOP` except that `M_STARTI` and `M_STOPI` are used to start and stop input.

M_UNHANGUP

Used to reconnect carrier after it has been dropped.

≡ *B*

Introduction

This appendix specifies the set of utility routines provided by STREAMS to assist development of modules and drivers.

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.

Note – The utility routines contained in this appendix represent an interface that will be maintained in subsequent versions of SunOS 5.x. Other than these utilities, functions in the STREAMS kernel code may change between versions.

Structure definitions are contained in Appendix A, “STREAMS Data Structures”. Routine references are found in this appendix. The following definitions are used:

Blocked

A queue that cannot be enabled due to flow control.

Enable

To schedule a queue’s `service` procedure to run.

Free

To release a STREAMS message or other data structure.

Message block (bp)

A triplet consisting of an `msgb` structure, a `datab` structure, and a data buffer. It is referenced by its type definition `mblk_t`.

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` structure).

Queue (q)

A queue structure. When it appears with “message” in certain utility description lines, it means “message queue”.

Schedule

To 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” generally refers to a module, driver, or Stream head.

Utility Descriptions

The STREAMS utility routines are described in the following section. A summary table is contained at the end of this appendix.

adjmsg – trim bytes in a message

```
int adjmsg(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 fails 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 and data block

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

`allocb()` returns a pointer to a message block of type `M_DATA`, in which the data buffer contains at least *size* bytes. *pri* is one of `BPRI_LO`, `BPRI_MED`, or `BPRI_HI` and indicates how critically the module needs the buffer. *pri* is currently unused and is maintained only for compatibility with applications developed prior to SunOS 5.3. If a block can not be allocated as requested, `allocb()` returns a `NULL` pointer.

When a message is allocated via `allocb()` the *b_band* field of the `mblk_t` is initially set to zero. Modules and drivers may set this field if necessary.

backq – get pointer to the queue behind a given queue

```
queue_t *backq(queue_t *cq);
```

`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` structure in Appendix A, “STREAMS Data Structures”) pointer is *q*. If no such queue exists (as when *q* is at a Stream end), `backq()` returns `NULL`.

bcanput – test for flow control in the given priority band

```
int bcanput(queue_t *q, unsigned char pri);
```

`bcanput()` provides modules and drivers with a way to test flow control in the given priority band. It returns 1 if a message of priority *pri* can be placed on the queue. It returns 0 if the priority band is flow controlled and sets the `QWANTW` flag for band zero (`QB_WANTW` for a nonzero band).

If no bands yet exist on the queue in question, 1 is returned. The call `bcanput(q, 0)` is equivalent to the call `canput(q)`.

bcanputnext – test for flow control in the given priority band

```
int bcanputnext(queue_t *q, unsigned char pri);
```

`bcanputnext()` provides modules and drivers with a way to test flow control in the given priority band for the queue pointed to by `q->q_next`. It returns 1 if the message of priority `pri` can be placed in the queue. It returns 0 if the priority band is flow controlled and sets the `QWANTW` flag for band zero (`QB_WANTW` for a nonzero band).

If the band does not yet exist on the queue in question, 1 is returned.

bufcall – recover from failure of allocb

```
int bufcall(unsigned int size,int pri,void (*func)(),long
arg, 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 is available. *pri* is as described in `allocb()`. When *func* is called, it has no user context 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 a non-zero id on success, indicating that the request has been successfully recorded, and 0 on failure. The returned id should be kept in the event `unbufcall()` needs to be called. 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(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 queued). 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, `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.

`canput()` only takes into account normal data flow control.

canputnext - test for room in the next queue

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

`Canputnext()` determines if there is room left in a message queue, pointed to by `q->q_next`. If the queue does not have a service procedure, `canputnext()` 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 queued.) If such a queue cannot be found, the search terminates on the queue at the end of the stream. `canputnext()` tests the queue found by the search. If the message queue in this queue is not full, `canputnext()` returns 1. This return indicates that a message can be put to queue `q`. If the message queue is full, `canputnext()` returns 0. In this case, the caller is generally referred to as blocked. `Canputnext()` only takes into account normal data flow control.

copyb - copy a message block

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

`copyb()` copies the contents of the message block pointed at by `bp` into a newly-allocated message block of at least the same size. `copyb()` allocates a new block by calling `allocb()`. 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(mblk_t *mp);
```

`copymsg()` uses `copyb()` to copy the message blocks contained in the message pointed at 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.

datamsq – test whether message is a data message

```
int datamsq(unsigned char type);
```

The `datamsq()` macro returns TRUE if `mp->b_datap->db_type` (where *mp* is declared as `mbblk_t *mp`) is a data type message (that is, not a control message). In this case, a data type is `M_DATA`, `M_PROTO`, `M_PCPROTO`, or `M_DELAY`. If `mp->b_datap->db_type` is any other message type, `datamsq()` returns FALSE.

dupb – duplicate a message block descriptor

```
mbblk_t *dupb(mbbk_t *bp);
```

`dupb()` duplicates the message block descriptor (`mbblk_t`) pointed at 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`) 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, `copymsg()` 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

```
mbblk_t *dupmsg(mbbk_t *mp);
```

`dupmsg()` calls `dupb()` to duplicate the message pointed at 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

```
void enableok(queue_t *q);
```

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

esballoc – allocate message and data blocks

```
mblk_t *esballoc(unsigned char *base, int size, int pri,  
frtn_t *fr_rtnp);
```

`esballoc()` allocates message and data blocks that point directly to a client-supplied buffer. `esballoc()` sets *db_base*, *b_rptr*, and *b_wptr* fields to *base* and *db_lim* to *base + size*. The pointer to struct `free_rtn` is placed in the *db_freep* field of the data block.

The method by which `free_func` is called is implementation specific. Do not assume that `free_func` will or will not be called directly from STREAMS utility routines like `freeb()`. The `free_func` function must not call another modules put procedure nor attempt to acquire a private module lock which may be held by another thread across a call to a STREAMS utility routine which could free a message block. Otherwise, the possibility for lock recursion and/or deadlock exists.

If an error occurs, `esballoc()` returns `NULL`.

Note – Modules and drivers using `esballoc` have to be prepared to handle calls to the `esballoc`'s callback `free` function after the last instance of the module or driver has been closed. In order to prevent the module text from being unloaded while there are outstanding `esballoc` callbacks, modules using `esballoc` should have their `_fini()` routine return `EBUSY`.

esbbcall – call function when buffer is available

```
int esbbcall(int pri, void (*func)(long arg), long arg);
```

`esbbcall()`, like `bufcall(9F)`, serves as a `timeout(9F)` call of indeterminate length. If `esballoc(9F)` is unable to allocate a message and data block header to go with its externally supplied data buffer, `esbbcall()` can be used to schedule the routine `func`, to be called with the argument `arg` when a buffer becomes available.

flushband – flush the messages in a given priority band

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

`flushband()` provides modules and drivers with the capability to flush the messages associated in a given priority band. *flag* is defined the same as in `flushq()`. If *pri* is zero, only ordinary messages are flushed. Otherwise, messages are flushed from the band specified by *pri* according to the value of *flag*.

If a queue behind *q* is blocked, `flushband()` may enable the blocked queue, as described in "getq – get a message from a queue"

flushq – flush a queue

```
void flushq(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`, `flushq()` discards all `M_DATA`, `M_PROTO`, `M_PCPROTO`, and `M_DELAY` messages, 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 "getq – get a message from a queue".

freeb – free a single message block

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

`freeb()` will free (deallocate) the message block descriptor pointed at by *bp*, and free the corresponding data block if the reference count (see "dupb – duplicate a message block descriptor") in the data block descriptor (data 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.

If the reference count is 1 and if the message was allocated by `esballoc()`, the function specified by the `db_frtnp->free_func` pointer is called with the parameter specified by `db_frtnp->free_arg`. `freeb()`. `freeb()` can't be used to free a multi-block message (see `freemsg()`).

Note – Results will be unpredictable if the `freeb()` is called with a null argument. You should always check that pointer is non-NULL before using `freeb()`.

freemsg – free all message blocks in a message

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

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

freezestr – freeze a stream

```
void freezestr(queue_t *q);
```

`freezestr()` freezes the state of the entire STREAM containing the queue *q*. A frozen STREAM blocks any thread attempting to enter any `open`, `close`, `put` or `service` routine belonging to any queue instance in the STREAM, and blocks any thread currently within the STREAM if it attempts to put messages onto or take messages off of any queue within the STREAM (with the sole exception of the caller). Threads blocked by this mechanism remain so until the STREAM is thawed by a call to `unfreezestr()`.

getq – get a message from a queue

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

`getq()` gets the next available message from the queue pointed at 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 marked with `QWANTR` so that the next time a message is placed on it, it will be scheduled for service (enabled, see `qenable()`). If the data in the queued messages in the queue drop 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 (that is, was blocked, see `canput()`), then the queue behind the current queue is scheduled for service.

The queue count is maintained on a per-band basis. Priority band 0 (normal messages) uses *q_count*, *q_lowat*, etc. Nonzero priority bands use the fields in their respective *qband* structures (*qb_count*, *qb_lowat*, etc.). All messages appear on the same list, linked via their *b_next* pointers.

q_count does not reflect the size of all messages on the queue; it only reflects those messages in the normal band of flow.

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

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

`insq()` places the message pointed at by *mp* in the message queue contained in the queue pointed at by *q* immediately before the already queued message pointed at 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.

If an attempt is made to insert a message out of order in a queue via `insq()`, the message will not be inserted and the routine fails.

The queue class of the new message is ignored. However, the priority band of the new message must adhere to the following ordering:

```
emp->b_prev->b_band >= mp->b_band >= emp->b_band.
```

This routine returns 1 on success and 0 on failure.

The stream must be frozen by the caller when calling `insq()`.

linkb – concatenate two messages into one

```
void linkb(mblk_t *mp, mblk_t *bp);
```

`linkb()` puts the message block pointed at by *bp* at the tail of the message pointed at by *mp*.

msgdsize – get the number of data bytes in a message

```
int msgdsize(mblk_t *mp);
```

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

msgpullup – concatenate bytes in a message

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

`msgpullup()` concatenates and aligns the first `len` data bytes of the message pointed to by *mp*, copying the data into a new message. Any remaining bytes in the remaining message blocks will be copied and linked onto the new message. The original message is unaltered.

noenable – prevent a queue from being scheduled

```
void noenable(queue_t *q);
```

`noenable()` prevents the queue *q* from being scheduled for service by `putq()` or `putbq()` when these routines queue an ordinary priority message, or by `insq()` when it queues any message. `noenable()` does not prevent the scheduling of queues when a high priority message is queued, unless it is queued by `insq()`.

OTHERQ – get pointer to the mate queue

```
queue_t *OTHERQ(queue_t *q);
```

The `OTHERQ()` function returns a pointer to the mate queue of *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 and align bytes in a message

```
int pullupmsg(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. `pullupmsg()` only concatenates across message blocks of similar type. It fails if *mp* points to a message of less than *len* bytes of similar type. If *len* is -1 `pullupmsg()` concatenates all the like-type blocks in the beginning of the message pointed at by *mp*.

On success, `pullupmsg()` returns 1 and, as a result of the concatenation, it may have altered the contents of the message pointed to by *mp*. On failure, it returns 0.

put – call a STREAMS put procedure

```
void put(queue_t *q, mblk_t *mp);
```

`put` calls the `put` procedure (`put(9E)` entry point) for the STREAMS queue specified by *q*, passing it the message block referred to by *mp*. It is typically used by a driver or module to call its own `put` procedure.

putbq – return a message to the beginning of a queue

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

`putbq()` puts the message pointed at by *mp* at the beginning of the queue pointed at by *q*, in a position in accordance with the message type. High priority messages are placed at the head of the queue, followed by priority band messages and ordinary messages. Ordinary messages are placed after all high priority and priority band messages, but before all other ordinary messages already in a queue. 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.

A service procedure must never put a high-priority message back on its own queue, as this would result in an infinite loop.

`putbq()` returns 1 on success and 0 on failure.

putctl – put a control message

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

`putctl()` creates a control message of type *type*, and calls the `put` procedure of the queue pointed at by *q*, with a pointer to the created message as an argument. `putctl()` allocates new blocks by calling `allocb()`.

On successful completion, `putctl()` returns 1. It returns 0, if it cannot allocate a message block, or if *type* `M_DATA`, `M_PROTO`, `M_PCPROTO`, or `M_DELAY` was specified.

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

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

`putctl1()` creates a control message of type *type* with a one-byte parameter *param*, and calls the `put` procedure of the queue pointed at by *q*, with a pointer to the created message as an argument. `putctl1()` allocates new blocks by calling `allocb()`.

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. `M_DELAY` is allowed.

putnext – put a message to the next queue

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

`putnext()` calls the `put` procedure of the next queue in a Stream and passes it a message pointer as an argument. *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.

putnextctl – put a control message

```
int putnextctl(queue_t *q, int type);
```

`putnextctl()` calls `putctl()`, passing the queue pointed at by *q*→*q_next*, and the message type *type*.

putnextctl1 – put a control message

```
int putnextctl1(queue_t *q, int type, int p);
```

`putnextctl()` calls `putctl()`, passing the queue pointed at by `q->q_next`, the message type `type` and the one byte parameter `param`.

putq – put a message on a queue

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

`putq()` puts the message pointed at by `mp` on the message queue contained in the queue pointed at by `q` and enables that queue. `putq()` queues messages based on message queueing priority.

The priority classes are high priority (`type >= QPCTL`), priority band (`type < QPCTL && band > 0`), and normal (`type < QPCTL && band == 0`).

`putq()` always enables the queue when a high-priority message is queued. `putq()` is allowed to enable the queue (`QNOENAB` is not set), if the message is the priority band message, or the `QWANTR` flag is set indicating that the service procedure is ready to read the queue.

Note – The service procedure must never put a priority message back on its own queue, as this would result in an infinite loop.

`putq()` enables 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, or if no message was found on the last `getq()` call, and no message has been queued since.

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

`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).

`putq()` returns 1 on success and 0 on failure.

qenable – enable a queue

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

`qenable()` places the queue pointed at by `q` on the linked list of queues that are ready to be called by the STREAMS scheduler.

qprocsoff – disable the put and service routines of a driver or module

```
void qprocsoff(queue_t *q);
```

`qprocsoff()` disables the put and service routines of the STREAMS driver or module. When the routines are disabled in a module, messages flow around the module as if it were not present in the stream. `qprocsoff()` must be called by the `close` routine of an MT-(Multi-thread) safe driver or module before releasing any resources on which the driver/module's put and service routines depend. `qprocsoff()` will remove the queue's service routines from the list of service routines to be run, and waits until any concurrent put or service routines are finished.

qprocson – enable the put and service routines of a driver or module

```
void qprocson(queue_t *q);
```

`qprocson()` enables the put and service routines of the STREAMS driver or module. Prior to the call to `qprocson()`, the put and service routines of a newly pushed module or newly opened driver are disabled. For the module, messages flow around it as if it were not present in the stream. `qprocson()` must be called by the first `open` of an MT-safe module or driver after allocation and initialization of any resource on which the put and service routines depend.

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

```
void qreply(queue_t *q, mblk_t *mp);
```

`qreply()` sends the message pointed at by *mp* up (or down) the Stream in the reverse direction from the queue pointed at 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.

qsize – find the number of messages on a queue

```
int qsize(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

```
queue_t * RD(queue_t *q);
```

`RD()` accepts either a read queue or write queue pointer, *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(mblk_t *mp, mblk_t *bp);
```

`rmvb()` removes the message block pointed at by *bp* from the message pointed at 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

```
void rmvq(queue_t *q, mblk_t *mp);
```

`rmvq()` removes the message pointed at by *mp* from the message queue in the queue pointed at 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.

The stream must be frozen by the caller when calling `rmvq()`.

strlog – submit messages for logging

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

`strlog()` submits messages containing specified information to the `log(7)` 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),
- `SL_NOTIFY` (request that a copy of the message be mailed to the system administrator)

Note – `SL_NOTIFY` is not an option by itself, but rather a modifier to the `SL_ERROR` flag

- `SL_CONSOLE` (log the message to the console),
- `SL_WARN` (warning message), and
- `SL_NOTE` (notice the message).

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 `log(7)`.)

strqget – obtain information about a queue or band of the queue

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

`strqget()` allows modules and drivers to get information about a queue or particular band of the queue. The information is returned in the *long* referenced by *valp*. The fields that can be obtained are defined by the following:

QHIWAT QLOWAT QMAXPSZ QMINPSZ QCOUNT QFIRST QLAST QFLAG QBAD

`strqget()` returns 0 on success and an error number on failure.

The stream must be frozen by the caller when calling `strqget()`.

strqset – change information about a queue or band of the queue

```
int strqset(queue_t *q, qfields_t what, unsigned char pri,
long val);
```

`strqset()` allows modules and drivers to change information about a queue or particular band (`pri`) of the queue. The updated information is provided by *val*. This routine returns 0 on success and an error number on failure. If the field is intended to be read-only, then the error `EPERM` is returned and the field is left unchanged.

See `<sys/stream.h>` for valid values of *what*.

The stream must be frozen by the caller when calling `strqset()`.

testb – check for an available buffer

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

`testb()` checks for the availability of a message buffer of size *size* 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 (for example, in the case of an interrupt routine taking buffers).

pri is as described in `allocb()`.

unbufcall – cancel a bufcall request

```
void unbufcall(int id);
```

`unbufcall()` cancels a `bufcall` request. *id* identifies an event in the `bufcall` request.



Caution – If a `bufcall()` request is made and the callback has not occurred before closing the driver, an `unbufcall()` must be made to cancel the scheduled callback. Otherwise a system crash may occur.

unfreezestr – unfreeze a stream

```
void unfreezestr(queue_t *q);
```

Unfreeze the entire STREAM pointed to by the queue *q*;

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

```
mblk_t *unlinkb(mblk_t *mp);
```

`unlinkb()` removes the first message block pointed at by *bp* 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

```
queue_t *WR(queue_t *q);
```

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

DKI Interface

With the DKI interface the following STREAMS utilities are implemented as functions: `datamsg`, `OTHERQ`, `putnext`, `RD`, `splstr`, and `WR`. `<sys/ddi.h>` must be included after `<sys/stream.h>` to get function definitions.

New MT perimeter utility routines

Note – The utility routines contained in this section represent facilities that are new to SunOS 5.3. These interfaces are subject to minor changes.

qbufcall – recover from failure of allocb

```
int qbufcall(queue_t *q, unsigned int size, int pri, void
(*func)(long arg), long arg);
```

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

`qbufcall()` arranges for `(*func)(arg)` to be called when a buffer of `size` bytes is available. `pri` is as described in `allocb()`. The framework enters the perimeters associated with the queue `q` prior to calling `func`. Thus the only difference between `qbufcall()` and `bufcall()` is that the former schedules a callback that is synchronous whereas the latter schedules an asynchronous callback. When `func` is called, it has no user context and must return without sleeping.

`qbufcall()` does not guarantee that the desired buffer will be available when `func` is called since interrupt processing may acquire it.

`qbufcall()` returns a non-zero id on success, indicating that the request has been successfully recorded, and zero on failure. The returned id should be kept in the event that `qunbufcall()` needs to be called. On a failure return, `func` will never be called. A failure indicates a (temporary) inability to allocate required internal data structures.

qtimeout – execute a function after a specified length of time

```
int qtimeout(queue_t *q, void(*ftn)(), caddr_t arg, long
ticks);
```

`qtimeout()` arranges for `(*func)(arg)` to be called after a specified time interval. The framework enters the perimeters associated with the queue `q` prior to calling `func`. Thus the only difference between `qtimeout()` and `timeout()` is that the former schedules a callback that is synchronous whereas the latter schedules an asynchronous callback. When `func` is called, it has no user context and must return without sleeping.

The exact time interval over which the timeout takes effect cannot be guaranteed, but the value is a close approximation.

`qtimeout()` returns a non-zero id on success, indicating that the request has been successfully recorded. Otherwise, if the `timeout()` table is full, the following panic message results: PANIC: Timeout table overflow.

The returned id should be kept in the event that `quntimeout()` needs to be *cal*anceled.

qunbufcall – cancel a qbufcall request

```
void qunbufcall(queue_t *q, int id);
```

`qunbufcall()` cancels a `qbufcall` request. *q* is the queue that was passed to `qbufcall` and *id* is the identifier returned by `qbufcall`.



Caution – If a `qbufcall()` request is made and the callback has not occurred before closing the driver, a `qunbufcall()` must be made to cancel the scheduled callback. Otherwise a system crash may occur.

quntimeout – cancel a qtimeout request

```
int quntimeout(queue_t *q, int id);
```

`quntimeout()` cancels a `qtimeout` request. *q* is the queue that was passed to `qtimeout` and *id* is the identifier returned by `qtimeout`.



Caution – If a `qtimeout()` request is made and the callback has not occurred before closing the driver, a `quntimeout()` must be made to cancel the scheduled callback. Otherwise a system crash may occur.

qwait/qwait_sig – STREAMS perimeter wait routines

```
void qwait(queue_t *q);
```

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

`qwait()` and `qwait_sig()` are used to wait for a message to arrive to the `put` or `service` procedures. They can be used only in the `open` and `close` procedures in a `STREAMS` module. `qwait()` and `qwait_sig()` atomically exit

the perimeters associated with the queue *q*, and wait for the next occurrence of a thread leaving the module's `put` or `service` procedures. Upon return `qwait()` and `qwait_sig()` re-enter the perimeters for the queue.

The difference between `qwait()` and `qwait_sig()` is that the latter can be interrupted by a signal. `qwait_sig()` normally returns non-zero, but when interrupted by a signal it returns zero.

`qwait()` and `qwait_sig()` serve the function, respectively, of `cv_wait()` and `cv_wait_sig()` for STREAMS modules that use perimeters.

qwriter - asynchronous STREAMS perimeter upgrade

```
void qwriter(queue_t *qp, mblk_t *mp, void(*func)(), int
perimeter);
```

`qwriter()` is used to upgrade the access at either the inner or the outer perimeter from shared to exclusive, when processing messages that require exclusive access in the `put` or `service` procedures.

`qwriter()` arranges for `(*func)(q, mp)` to be called when exclusive access has been granted at the specified *perimeter*.

`qwriter()` will be upgraded to exclusive access immediately if possible, in which case *func* will be called before `qwriter()` returns. If it is not possible to upgrade without blocking, `qwriter()` will defer the upgrade until later and return before *func* has executed. Thus modules cannot assume that *func* has executed when `qwriter()` returns. One way to avoid dependencies on this is for the module to immediately return after calling `qwriter()` and let *func* perform all the remaining processing of the message *mp*.

When `qwriter()` defers executing *func*, the STREAMS framework will prevent other messages from entering the inner perimeter associated with the queue until the asynchronous upgrade has completed and *func* has finished executing.

Utility Routine Summary

Table C-1 Summary of Utility Routines

ROUTINE	DESCRIPTION
adjmsg	trim bytes in a message
allocb	allocate a message block
backq	get pointer to the queue behind a given queue
bcanput	test for flow control in a given priority band
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
esballoc	allocate message and data blocks
flushband	flush messages in a given priority band
flushq	flush a queue
freeb	free a message block
freemsg	free all message blocks in a message
freezestr	disable changes to the state of the stream
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 and align bytes in a message

Table C-1 Summary of Utility Routines

ROUTINE	DESCRIPTION
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
qbufcall	call a function when a buffer becomes available
qprocsoff	turn off queue processing
qprocson	turn on queue processing
qreply	send a message on a Stream in the reverse direction
qsize	find the number of messages on a queue
qtimeout	execute a function after a specified length of time
qunbufcall	cancel a pending qbufcall request
quntimeout	cancel a pending qtimeout request
qwait	perimeter wait routine
qwait_sig	perimeter wait routine
qwriter	asynchronous perimeter upgrade
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
strqget	obtain information on a queue or a band of the queue
strqset	change information on a queue or a band of the queue
testb	check for an available buffer
unbufcall	cancel bufcall request

Table C-1 Summary of Utility Routines

ROUTINE	DESCRIPTION
unfreezestr	enable changes to the state of the stream
unlinkb	remove a message block from the head of a message
WR	get pointer to the write queue

Overview of Debugging Facilities

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

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

1. Kernel debug printing

This consists of kernel facilities for printing from inside drivers

2. STREAMS error logging

This is a STREAMS-supported model of generating error messages and allowing them to be received by one of three different types of loggers.

3. Kernel-examination tools

These include the tools bundled with the operating system

Kernel Debug Printing

Console Messages

The kernel routine `cmn_err()` allows printing of formatted strings on a system console. It displays a specified message on the console and/or stores it in the `putbuf` that is a circular array in the kernel and contains output from `cmn_err()`. Its format is:

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

void cmn_err (int level, char *fmt, int ARGS)
```

where *level* can take the following values:

- `CE_CONT` - may be used as simple `printf()`. It is used to continue another message or to display an informative message not associated with an error.
- `CE_NOTE` - report system events. It is used to display a message preceded with `NOTICE:`. This message is used to report system events that do not necessarily require user action, but may interest the system administrator. For example, a sector on a disk needing to be accessed repeatedly before it can be accessed correctly might be such an event.
- `CE_WARN` - system events that require user action. This is used to display a message preceded with `WARNING:`. This message is used to report system events that require immediate attention, such as those where if an action is not taken, the system may panic. For example, when a peripheral device does not initialize correctly, this level should be used.
- `CE_PANIC` - system panic. This is used to display a message preceded with `PANIC:`. Drivers should specify this level only under the most severe conditions. A valid use of this level is when the system cannot continue to function. If the error is recoverable, not essential to continued system operation, do not panic the system. This level halts all processing.

fmt and *ARGS* are passed to the kernel routine `printf()` that runs at `splhi()` and should be used sparingly. If the first character of *fmt* begins with `!` (an exclamation point) output is directed to `putbuf`. `putbuf` can be accessed with the `crash(1M)` command. If the destination character begins with `^` (a caret) output goes to the console. If no destination character is specified, the message is directed to both the `putbuf` array and the console.

`cmn_err()` appends each *fmt* with `"\n"`, except for the `CE_CONT` level, even when a message is sent to the `putbuf` array. *ARGS* specifies a set arguments passed when the message is displayed. Valid specifications are `%s` (string), `%u` (unsigned decimal), `%d` (decimal), `%o` (octal), and `%x` (hexadecimal). `cmn_err()` does not accept length specifications in conversion specifications. For example, `%3d` is ignored.

STREAMS Error Logging

Error and Trace Logging

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

Any module or driver in any Stream can call the STREAMS logging function `strlog`, described in `log(7)`. `strlog` is also described in Appendix C. When called, `strlog` will send formatted text to the error logger `strerr(1M)`, the trace logger `strace(1M)`, or the console logger.

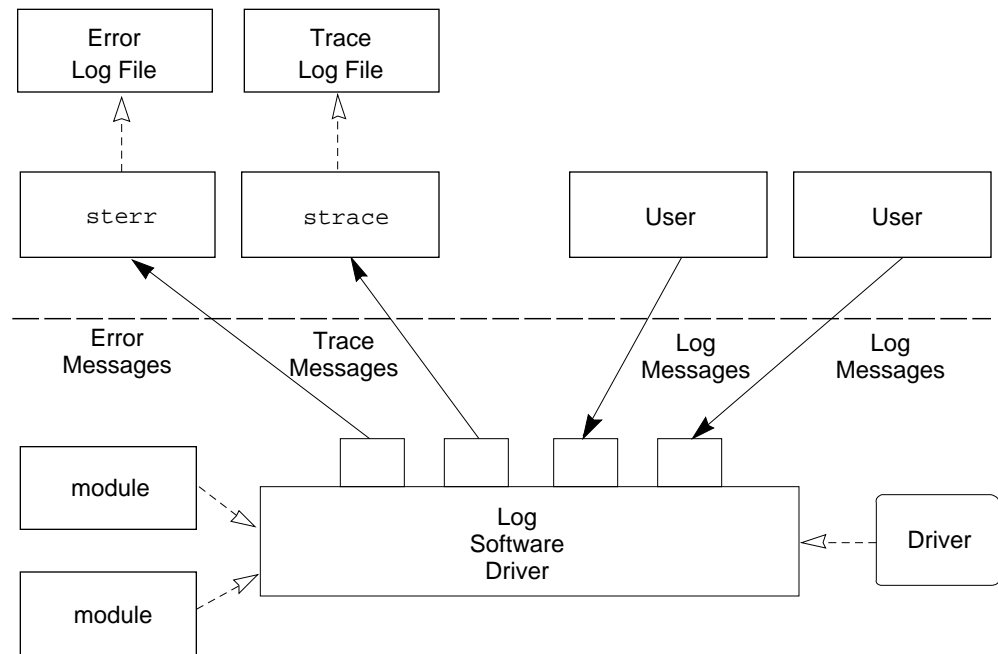


Figure D-1 Error and Trace Logging

`sterr` is intended to operate as a daemon process initiated at system startup. A call to `strlog` requesting an error to be logged causes an `M_PROTO` message to be sent to `sterr`, which formats the contents and places them in a daily file. The utility `strclean(1M)` is provided to purge daily log files that have not been modified for three days.

A call to `strlog` requesting trace information to be logged causes a similar `M_PROTO` message to be sent to `strace(1M)`, which places it in a user designated file. `strace` is intended to be initiated by a user. The user can designate the modules/drivers and severity level of the messages to be accepted for logging by `strace`.

A user process can submit its own `M_PROTO` messages to the log driver for inclusion in the logger of its choice through `putmsg(2)`. The messages must be in the same format required by the logging processes and will be switched to the logger(s) requested in the message.

The output to the log files is formatted, ASCII text. The files can be processed by standard system commands such as `grep(1)` or `ed(1)`, or by developer-provided routines.

Kernel Examination Tools

The crash(1M) Command

The `crash` command is used to examine kernel structures interactively. It can be used on a system dump and on an active system.

The following `crash` functions are related to STREAMS:

- `dbfreePrint` data block header free list
- `dblockPrint` allocated STREAMS data block headers
- `linkblkPrint` the `linkblk` table
- `mbfreePrint` free STREAMS message block headers
- `mblockPrint` allocated STREAMS message block headers
- `ptyPrint` pseudo ttys now configured. The `-l` option gives information on the line discipline module `ldterm`, the `-h` option provides information on the pseudo-tty emulation module `ptem`, and the `-s` option gives information on the packet module `pckt`.
- `qrunPrint` a list of scheduled queues
- `queuePrint` the STREAMS queues
- `streamPrint` the `stdat` table
- `strstatPrint` STREAMS statistics
- `ttyPrint` the `tty` table. The `-l` option prints details about the line discipline module.

The `crash` functions `dblock`, `linkblk`, `mblock`, `queue`, and `stream` take an optional table entry argument or address that is the address of the data structure. The `strstat` command gives information about STREAMS event cells and `linkblks` in addition to message blocks, data blocks, queues, and

streams. On the output report, the `CONFIG` column represents the number of structures currently configured. It may change because resources are allocated as needed.

The `adb(1)` Command

`adb` is an interactive general-purpose debugger. It can be used to examine files and provides a controlled environment for the execution of programs. It has no support built in for any STREAMS functionality.

The `kadb(1M)` Command

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

Configuration



Introduction

This appendix contains information about configuring STREAMS drivers and modules into SunOS 5.3. This chapter discusses how to configure a driver and a module for the STREAMS framework only. For more in-depth information on the general configuration mechanism, see *Writing Device Drivers*.

This appendix also includes a list of STREAMS related tunable parameters and describes the autopush facility.

Configuring STREAMS Drivers and Modules

The following sections contain descriptions of the pointer relationships maintained by the kernel and the various data structures used in STREAMS drivers. For the kernel to access a driver, it uses a sequence of pointers in various data structures. Look first at the data structure relationship and then the entry point interface for loading the driver into the kernel and accessing the driver from the application level.

The order of data structure traversal the kernel uses to get to a driver is as follows:

1. `modlinkage` - contains the revision number and a list of drivers to dynamically load. It is used by `mod_install` in the `_init()` routine to load the module into the kernel. Points to a `modldrv` or `modlstrmod`.

2. `modldrv` - contains info about the driver being loaded, points to the `devops` structure
3. `modlstrmod` - points to an `fmodsw` structure (which points to a `streamtab`) Only used by STREAMS modules.
4. `devops` - contains list of entry points for a driver, such as `identify`, `attach`, and `info`. Also points to a `cb_ops` structure.
5. `cb_ops` - points to list of threadable entry points to driver, like `open`, `close`, `read`, `write`, `ioctl`. Also points to the `streamtab`
6. `streamtab` - points to the read and write queue init structures
7. `qinit` - points to the entry points of the STREAMS portion of the driver, such as `put`, `srv`, `open`, `close`, as well as the `mod_info` structure. These entry points only process messages.

Each STREAMS driver or module contains the linkage connections for the various data structures. This contains a list of pointers to `dev_ops` structures. Within each `dev_ops` structure, there is a pointer to the `cb_ops` structure. Within the `cb_ops` structure, there is a pointer named `streamtab`. If the driver in question is *not* a STREAMS driver, then `streamtab` will be `NULL`. If the driver *is* a STREAMS driver, then `streamtab` will point to a structure that contains initialization routines for the driver.

Data Structure Layout

Now look at the structures presented above and see what they consist of and how they fit together. Starting with the `modlinkage` structure, we will work through the chain of pointers connecting all the different data structures. We will then look at the *loadable module wrappers* (for regular device drivers) and STREAMS module wrappers.

modlinkage

This is the definition of `modlinkage`. See `modlinkage(9S)`.

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

```

* linkage structures */
};

```

modldrv

This is the definition of `modldrv(9S)`.

```

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

```

modlstrmod

This is the definition of `modlstrmod(9S)`. It does not access `devops` structures because modules can only be pushed onto an existing stream.

```

struct modlstrmod {
    struct mod_ops      *strmod_modops;
    char                *strmod_linkinfo;
    struct fmodsw       *strmod_fmodsw;
};

```

dev_ops

The first structure is `dev_ops(9S)`. It represents a specific class or type of device. Each `dev_ops` structure represents a unique device to the operating system. Each device has its own `dev_ops` structure. Within each `dev_ops` structure is a `cb_ops`.

```

struct dev_ops {
    int     devo_rev;           /* Driver build version*/
    int     devo_refcnt;       /* device reference count*/
};

```

```
int      (*devo_getinfo)(dev_info_t *dip,
                        ddi_info_cmd_t infocmd, void *arg, void **result);
int      (*devo_identify)(dev_info_t *dip);
int      (*devo_probe)(dev_info_t *dip);
int      (*devo_attach)(dev_info_t *dip, ddi_attach_cmd_t cmd);
int      (*devo_detach)(dev_info_t *dip, ddi_detach_cmd_t cmd);
int      (*devo_reset)(dev_info_t *dip, ddi_reset_cmd_t cmd);

struct cb_ops*devo_cb_ops; /* cb_ops ptr for leaf driver*/
struct bus_ops  *devo_bus_ops; /* ptr for nexus drivers */
};
```

cb_ops

The *cb_ops*(9S) structure is the SunOS 5.x version of the *cdevsw* and *bdevsw* tables from previous versions of System V. It contains character and block device information. It contains the driver entry points for non-STREAMS drivers.

```
struct cb_ops {
int      *cb_open)(dev_t *devp, int flag, int otyp, cred_t *credp);
int      (*cb_close)(dev_t dev, int flag, int otyp, cred_t *credp);
int      (*cb_strategy)(struct buf *bp);
int      (*cb_print)(dev_t dev, char *str);
int      (*cb_dump)(dev_t dev, caddr_t addr, daddr_t blkno, int nblk);
int      (*cb_read)(dev_t dev, struct uio *uiop, cred_t *credp);
int      (*cb_write)(dev_t dev, struct uio *uiop, cred_t *credp);
int      (*cb_ioctl)(dev_t dev, int cmd, int arg, int mode,
                    cred_t *credp, int *rvalp);
int      (*cb_devmap)(dev_t dev, dev_info_t *dip,
                    ddi_devmap_data_t *dvdp, ddi_devmap_cmd_t cmd,
                    off_t offset, unsigned int len, unsigned int prot,
                    cred_t *credp);
int      (*cb_mmap)(dev_t dev, off_t off, int prot);
int      (*cb_segmap)(dev_t dev, off_t off, struct as *asp,
                    caddr_t *addrp, off_t len, unsigned int prot,
                    unsigned int maxprot, unsigned int flags, cred_t *credp);
int      (*cb_chpoll)(dev_t dev, short events, int anyyet,
                    short *reventsp, struct pollhead **phpp);
int      (*cb_prop_op)(dev_t dev, dev_info_t *dip,
```

```

        ddi_prop_op_t prop_op, int mod_flags,
        char *name, caddr_t valuep, int *length);

    struct streamtab *cb_str;           /* streams information */

    /*
     * The cb_flag fields are here to tell the system a
     * bit about the device. The bit definitions are
     * in <sys/conf.h>.
     */
    int      cb_flag;                  /* driver compatability flag */
};

```

streamtab

The `streamtab` structure contains pointers to the structures that hold the routines that actually initialize the reading and writing for module. These definitions are also in Appendix A, “STREAMS Data Structures”.

If `streamtab` is `NULL`, then it signifies no STREAMS routines and the entire driver is treated as though it were a regular driver. The `streamtab` is used to indirectly identify the appropriate open, close, put, service, and administration routines. These driver and module routines should generally be declared `static`.

```

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

```

qinit

The `qinit` structure (also shown in Appendix A) contains pointers to the STREAMS entry points. These routines are called by the module loading code in the kernel.

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

Entry Points

As described in Chapter 9, "Drivers", and as also seen in the previous data structures, there are four types of entry points:

1. Kernel module loading - `_init(9E)`, `_fini(9E)`, `_info(9E)`
2. `dev_ops` - `identify(9E)`, `attach(9E)`, `getinfo(9E)`.
3. `cb_ops` - `xxopen()`, `xxclose()`, `xxread()`, `xxwrite()`, `xxioctl()`.
4. `streamtab` - `xxput()`, `xxsrv()`.

pts example

Now look at a real example taken from SunOS 5.3. This is the driver `pts`, which is the pseudo terminal slave driver.

```
/*
 * Slave Stream Pseudo Terminal Module
 */

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



```
include <sys/stream.h>
include <sys/stropts.h>
include <sys/stat.h>
include <sys/errno.h>
include <sys/debug.h>
include <sys/cmn_err.h>
include <sys/modctl.h>
include <sys/conf.h>
include <sys/ddi.h>
inclde <sys/sunddi.h>

static int ptsopen (queue_t*, dev_t*, int, int, cred_t*static int
ptsclose (queue_t*, int, cred_t*);
static int ptswput (queue_t*, mblk_t*);
static int ptsrsrv (queue_t*);
static int ptswsrv (queue_t*);

static int pts_devinfo(dev_info_t *dip, ddi_info_cmd_t infocmd,
void *arg,void **result);

static struct module_info pts_info = {
    0xface,
    "pts",
    0,
    512,
    512,
    128
};

static struct qinit ptsrint = {
    NULL,
    ptsrsrv,
    ptsopen,
    ptsclose,
    NULL,
    &pts_info,
    NULL
};

static struct qinit ptswint = {
    ptswput,
    ptswsrv,
    NULL,
    NULL,
    NULL,
    NULL,
```

```

        &pts_info,
        NULL
    };

    static struct streamtab ptsinfo = {
        &ptsrint,
        &ptswint,
        NULL,
        NULL
    };

    static int pts_identify(dev_info_t *devi);
    static int pts_attach(dev_info_t *devi, ddi_attach_cmd_t cmd);
    static int pts_detach(dev_info_t *devi, ddi_detach_cmd_t cmd);
    static dev_info_t *pts_dip; /* private copy of
    devinfo ptr */

    extern kmutex_t pt_lock;
    extern pt_cnt;
    static struct          cb_ops  cb_pts_ops = {
        nulldev,          /* cb_open */
        nulldev,          /* cb_close */
        nodev,            /* cb_strategy */
        nodev,            /* cb_print */
        nodev,            /* cb_dump */
        nodev,            /* cb_read */
        nodev,            /* cb_write */
        nodev,            /* cb_ioctl */
        nodev,            /* cb_devmap */
        nodev,            /* cb_mmap */
        nodev,            /* cb_segmap */
        nochpoll,         /* cb_chpoll */
        ddi_prop_op,      /* cb_prop_op */
        &ptsinfo,         /* cb_stream */
        D_MP               /* cb_flag */
    };

    static struct          dev_ops  pts_ops = {
        DEVO_REV,         /* devo_rev */
        0,                /* devo_refcnt */
        pts_devinfo,      /* devo_getinfo */
        pts_identify,     /* devo_identify */
        nulldev,          /* devo_probe */
        pts_attach,       /* devo_attach */
        pts_detach,       /* devo_detach */
    };

```

```

        nodev,          /* devo_reset */
        &cb_pts_ops,     /* devo_cb_ops */
        (struct bus_ops*) NULL /* devo_bus_ops */
};

/*
 * Module linkage information for the kernel.
 */

static struct modldrv modldrv = {
    &mod_driverops, /* Type of module: a pseudo driver */
    "Slave Stream Pseudo Terminal driver'pts'",
    &pts_ops, /* driver ops */
};

static struct modlinkage modlinkage = {
    MODREV_1,
    (void *)&modldrv,
    NULL
};

int
_init(void)
{
    return (mod_install(&modlinkage));
}

int
_fini(void)
{
    return (mod_remove(&modlinkage));
}

int
_info(struct modinfo *modinfop)
{
    return (mod_info(&modlinkage, modinfop));
}

static int
pts_identify(dev_info_t *devi)
{
    if (strcmp(ddi_get_name(devi), "pts") == 0)
        return (DDI_IDENTIFIED);
    else

```

```

        return (DDI_NOT_IDENTIFIED);
    }

    static int
    pts_attach(dev_info_t *devi, ddi_attach_cmd_t cmd)
    {
        int i;
        char name[5];

        if (cmd != DDI_ATTACH)
            return (DDI_FAILURE);

        for (i = 0; i < pt_cnt; i++) {
            (void) sprintf(name, "%d", i);
            if (ddi_create_minor_node(devi, name, S_IFCHR,
                i, NULL, 0) == DDI_FAILURE) {
                ddi_remove_minor_node(devi, NULL);
                return (DDI_FAILURE);
            }
        }
        return (DDI_SUCCESS);
    }

    static int
    pts_detach(dev_info_t *devi, ddi_detach_cmd_t cmd)
    {
        ddi_remove_minor_node(devi, NULL);
        return (DDI_SUCCESS);
    }

    static int
    pts_devinfo (dev_info_t *dip, ddi_info_cmd_t infocmd, void *arg,
        void **result)
    {
        int error;

        switch (infocmd) {
            case DDI_INFO_DEVT2DEVINFO:
                if (pts_dip == NULL) {
                    error = DDI_FAILURE;
                } else {
                    *result = (void *) pts_dip;
                    error = DDI_SUCCESS;
                }
                break;
        }
    }

```

```

        case DDI_INFO_DEVT2INSTANCE:
            *result = (void *) 0;
            error = DDI_SUCCESS;
            break;
        default:
            error = DDI_FAILURE;
    }
    return (error);
}

/* the open, close, wput, rsrv, and wsrsv routines are presented
 * here solely for the sake of showing how they interact with the
 * configuration data structures and routines. Therefore, the
 * bulk of their code is not included.
 */
static int
ptsopen(rqp, devp, oflag, sflag, credp)
    queue_t *rqp; /* pointer to the read side queue */
    dev_t *devp; /* pointer to stream tail's dev */
    int oflag; /* the user open(2) supplied flags
 */
    int sflag; /* open state flag */
    cred_t *credp; /* credentials */
{
    qprocson(rqp);
    return (0);
}

static int
ptsclose(rqp, flag, credp)
    queue_t *rqp;
    int flag;
    cred_t *credp;
{
    qprocsoff(rqp);
    return (0);
}

static int
ptswput(qp, mp)
    queue_t *qp;
    mblk_t *mp;
{
    return (0);
}

```

```
static int
ptsrsrv(qp)
    queue_t *qp;
{
    return (0);
}

static int
ptswsrv(qp)
    queue_t *qp;
{
    return (0);
}
```

STREAMS Module Configuration

Here are the structures if you are working with a module instead of a driver. Notice that it is an `modlstrmod` used in `modlinkage` and `fmodsw` points to `streamtab` instead of going through `devops`.

```
extern struct streamtab pteminfo;

static struct fmodsw fsw = {
    "ptem",
    &ptemininfo,
    D_NEW | D_MP
};

/*
 * Module linkage information for the kernel.
 */
extern struct mod_ops mod_strmodops;

static struct modlstrmod modlstrmod = {
    &mod_strmodops, "pty hardware emulator", &fsw
};

static struct modlinkage modlinkage = {
    MODREV_1, (void *)&modlstrmod, NULL
};
```

Compilation

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

```
cc -D_KERNEL -c example_one.c
cc -D_KERNEL -c example_two.c
as -P -D_ASM -D_KERNEL -I. -o example_asm.o example_asm.s
ld -r -o example example_one.o example_two.o example_asm.o
```

Kernel Loading

See *Writing Device Drivers* for more information on the sequence of installing and loading device drivers. The procedures are to copy your driver or module to /kernel/drv or /kernel/strmod, respectively, and for drivers run add_drv(1M).

Checking module type

Next, see the code that enables a driver to determine if it is running as a regular driver, a module, or a cloneable driver. The open routine returns the *sflag* which is checked.

```
if (sflag == MODOPEN)
    /* then the module is being pushed */
else if (sflag == CLONEOPEN)
    /* then its being opened as a cloneable driver */
else
    /* its being opened as a regular driver */
```

Tunable Parameters

Certain system parameters referenced by STREAMS are configurable when building a new operating system (see the file /etc/system and the *SunOS User's Guide to System Administration* for further details). These parameters are:

nstrpush

Maximum number (should be at least 8) of modules that may be pushed onto a single Stream.

strmsgsz

Maximum number of bytes of information that a single system call can pass to a Stream to be placed into the data part of a message (in `M_DATA` blocks). Any `write(2)` exceeding this size will be broken into multiple messages. A `putmsg(2)` with a data part exceeding this size will fail with `ERANGE`. If `STRMSGSZ` is set to 0, then the number of bytes passed to a Stream is effectively infinite.

strctlsz

Maximum number of bytes of information that a single system call can pass to a Stream to be placed into the control part of a message (in an `M_PROTO` or `M_PCPROTO` block). A `putmsg(2)` with a control part exceeding this size will fail with `ERANGE`.

Autopush Facility

The Autopush facility (see `autopush(1M)`) is a general mechanism that configures the list of modules for a STREAMS device. It automatically pushes a pre-specified list (`/etc/iu.ap`) of modules onto the Stream when the STREAMS device is opened and the device is not already open.

The STREAMS Administrative Driver (SAD) (see `sad(7)`) provides an interface to the autopush mechanism. System administrators can open the SAD driver and set or get autopush information on other drivers. The SAD driver caches the list of modules to push for each driver. When the driver is opened, if not already open, the Stream head checks the SAD's cache to see if the device opened has been configured to have modules pushed automatically. If an entry is found, the modules are pushed. If the device has already been opened but has not yet been closed, another open would not cause the list of the pre-specified modules to be pushed again.

Three options are available to configure the module list:

- Configure for each minor device - that is, a specific major and minor device number.
- Configure for a range of minor devices within a major device.
- Configure for all minor devices within a major device.

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

User Interface

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

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

The `ioctl` is a form of `ioctl(fd, cmd, arg)`, where `fd` is the file descriptor of the SAD driver, `cmd` is either `SAD_SAP` (set autopush information) or `SAD_GAP` (get autopush information), and `arg` is a pointer to the structure `strapush`.

The structure `strapush` is defined as:

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

/* autopush information common to user and kernel */

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

```

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

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

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

```

A device is identified by its major device number, *sap_major*. The SAD_SAP ioctl (*sap_cmd*) can take the following options:

- SAP_ONE configures a single minor device, *sap_minor*, of a driver.
- SAP_RANGE configures a range of minor devices from *sap_minor* to *sap_lastminor*, inclusive.
- SAP_ALL configures all minor devices of a device.
- SAP_CLEAR clears the previous settings by removing the entry with the matching *sap_major* and *sap_minor* fields.

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

A user may query the current configuration status of a given major/minor device by issuing the SAD_GAP ioctl with *sap_major* and *sap_minor* values of the device set. On successful return from this system call, the *strapush*

structure will be filled in with the corresponding information for that device. The maximum number of entries the SAD driver can cache is determined by the tunable parameter `NAUTOPUSH` found in the SAD driver's master file.

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

```
# /dev/console and /dev/contty autopush setup
#
#   major      minor      lastminor    modules
wc          0           0           ldterm ttcompat
zs          0           1           ldterm ttcompat
ptsl        0           15          ldterm ttcompat
```

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

Manual Pages



This appendix lists the STREAMS related manual pages. The manual pages are supplied in the SunOS 5.3 Reference Manuals and are listed here for reference.

- `strchg(1)` - change or query Stream configuration
- `autopush(1M)` - configure automatically pushed STREAMS modules
- `fdetach(1M)` - detach a name from a STREAMS-based file descriptor
- `strace(1M)` - print STREAMS trace messages
- `strclean(1M)` - STREAMS error logger cleanup program
- `strerr(1M)` - STREAMS error logger daemon
- `close(2)` - close a file descriptor
- `getmsg(2)` - get next message off a Stream
- `ioctl(2)` - control device
- `open(2)` - open for reading or writing
- `poll(2)` - STREAMS input/output multiplexing
- `putmsg(2)` - send a message on a Stream
- `read(2)` - read from file
- `write(2)` - write in a file
- `fattach(3C)` - attach a STREAMS file descriptor to a filename
- `fdetach(3C)` - disassociate a file name from a named Stream

- `grantpt(3C)` - grant access to the slave pseudo-terminal device
- `iastream(3C)` - determine if `fd` is associated with a STREAMS device
- `ptsname(3C)` - get the slave pseudo-terminal device name
- `unlockpt(3C)` - unlock a pseudo-terminal master/slave pair
- `audio(7)` - telephone-quality audio device
- `clone(7)` - open a major/minor device a STREAMS driver
- `connld(7)` - gain a unique, non-multiplexed connection to a server
- `console(7)` - STREAMS based console interface
- `ldterm(7)` - standard STREAMS terminal line discipline module
- `log(7)` - interface to STREAMS error logging and event tracing
- `pckt(7)` - push a PCKT module (packet mode) on the master side
- `ptem(7)` - process terminal `ioctl` messages
- `sad(7)` - STREAMS Administrative Driver
- `streamio(7)` - STREAMS `ioctl` commands
- `timod(7)` - Transport Interface cooperating STREAMS module
- `tirdwr(7)` - Transport Interface read/write interface STREAMS module
- `ttcompat(7)` - V7, 4BSD and Xenix STREAMS compatibility module
- `zs(7)` - Zilog 8530 SCC serial communications driver

DDI/DKI Entry Points

- `close(9E)` - relinquish access to a device
- `fini` - prepare a loadable module for loading
- `info` - provide information about a loadable module
- `init(9E)` - initialize a loadable module
- `open(9E)` - gain access to a device
- `put(9E)` - receive messages from the preceding queue
- `srv(9E)` - service queued messages

DDI/DKI Functions

- `OTHERQ(9F)` - get pointer to queue's partner queue
- `RD(9F)` - get pointer to the read queue
- `SAMESTR(9F)` - test if next queue is in the same stream
- `WR(9F)` - get pointer to the write queue for this module or driver
- `adjmsg(9F)` - trim bytes from a message
- `allocb(9F)` - allocate a message block
- `backq(9F)` - get pointer to the queue behind the current queue
- `bcanput(9F)` - test for flow control in specified priority band
- `bufcall(9F)` - call a function when a buffer becomes available
- `canput(9F)` - test for room in a message queue
- `canputnext(9F)` - test for room in a message queue
- `copyb(9F)` - copy a message block
- `copymsg(9F)` - copy a message
- `datamsg(9F)` - test whether a message is a date message
- `dupb(9F)` - duplicate a message block descriptor
- `dupmsg(9F)` - duplicate a message
- `enableok(9F)` - reschedule a queue for service
- `esballoc(9F)` - allocate a message block using a caller-supplied buffer
- `esbbcall(9F)` - call function when buffer is available
- `flushband(9F)` - flush messages for a specified priority band
- `flushq(9F)` - remove messages from a queue
- `freeb(9F)` - free a message block
- `freemsg(9F)` - free all message blocks in a message
- `freezestr(9F)` - freeze and unfreeze the state of a stream
- `getq(9F)` - get the next message from a queue
- `insq(9F)` - insert a message into a queue

- `linkb(9F)` - concatenate two message blocks
- `msgdsize(9F)` - return the number of bytes in a message
- `noenable(9F)` - prevent a queue from being scheduled
- `otherq(9F)` - get pointer to queue's partner queue
- `pullupmsg(9F)` - concatenate bytes in a message
- `putbq(9F)` - place a message at the head of a queue
- `putctl(9F)` - send a control message to a queue
- `putctl1(9F)` - send a control message with a one-byte parameter to a queue
- `putnext(9F)` - send a message to the next queue
- `putnextctl(9F)` - send a control message to a queue
- `putq(9F)` - put a message on a queue
- `qenable(9F)` - enable a queue
- `qprocsoff(9F)/qprocson(9F)` - disable/enable STREAMS put and service procedures
- `qreply(9F)` - send a message on a stream in the reverse direction
- `qsize(9F)` - find the number of messages on a queue
- `rd(9F)` - get pointer to the read queue
- `rmvb(9F)` - remove a message block from a message
- `rmvq(9F)` - remove a message from a queue
- `samestr(9F)` - test if next queue is in the same stream
- `strlog(9F)` - submit messages to the log driver
- `strqget(9F)` - get information about a queue or band of the queue
- `strqset(9F)` - permits updating the value of certain streams' queues parameters
- `testb(9F)` - check for an available buffer
- `unlinkb(9F)` - remove a message block from the head of a message
- `wr(9F)` - get pointer to the write queue for this module or driver

DDI/DKI Data Structures

- `List(9S)` - List of structures
- `datab(9S)` - STREAMS message data structure
- `free_rtn(9S)` - structure that specifies a driver's message freeing routine
- `module_info(9S)`- STREAMS driver identification and limit value structure
- `msgb(9S)` - STREAMS message block structure
- `qband(9S)` - STREAMS queue flow control information structure
- `qinit(9S)` - STREAMS queue processing procedures structure
- `queue(9S)` - STREAMS queue structure
- `streamtab(9S)` - STREAMS entity declaration structure

Glossary



autopush

A STREAMS mechanism that enables a pre-specified list of modules to be pushed automatically onto the Stream when a STREAMS device is opened. This mechanism is used only for administrative purposes.

back-enable

To enable (by STREAMS) a preceding blocked queue's `service` procedure when STREAMS determines that a succeeding queue has reached its low watermark.

blocked

A queue's `service` procedure that cannot be enabled due to flow control.

clone device

A STREAMS device that returns an unused major/minor device when initially opened, rather than requiring the minor device to be specified by name in the `open(2)` call.

close routine

A procedure that is called when a module is popped from a Stream or when a driver is closed.

**controlling Stream**

A Stream above the multiplexing driver used to establish the lower connections. Multiplexed Stream configurations are maintained through the controlling Stream to a multiplexing driver.

DDI

Device Driver Interface. An interface that facilitates driver portability across different UNIX system versions on SPARC® hardware.

DKI

Driver–Kernel Interface. An interface between the UNIX system kernel and different types of drivers. It consists of a set of driver-defined functions that are called by the kernel. These functions are entry points into a driver.

downstream

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

device driver

A Stream component whose principle functions are handling an associated physical device and transforming data and information between the external interface and Stream.

driver

A module that forms the Stream end. It can be a device driver or a pseudo-device driver. It is a required component in STREAMS (except in STREAMS-based pipe mechanism), and physically identical to a module. It typically handles data transfer between the kernel and a device and does little or no processing of data.

enable

A term used to describe scheduling of a queue's `service` procedure.

FIFO

First-In-First-Out. A term for named pipes. This term is also used in queue scheduling.



flow control

A STREAMS mechanism that regulates the rate of message transfer within a Stream and from user space into a Stream.

hardware emulation module

A module required when the terminal line discipline is on a Stream but there is no terminal driver at the end of a Stream. This module understands all `ioctl`s necessary to support terminal semantics specified by `termio(7)` and `termios(7)`.

input side

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

line discipline

A STREAMS module that performs `termio(7)` canonical and non-canonical processing. It shares some `termio(7)` processing with a driver in a STREAMS terminal subsystem.

lower Stream

A Stream connected below a multiplexer pseudo-device driver, by means of an `I_LINK` or `I_PLINK` `ioctl`. The far end of a lower Stream terminates at a device driver or another multiplexer driver.

master driver

A STREAMS-based device supported by the pseudo-terminal subsystem. It is the controlling part of the pseudo-terminal subsystem (also called `ptm`).

message

One or more linked message blocks. A message is referenced by its first message block and its type is defined by the message type of that block.

message block

A triplet consisting of a data buffer and associated control structures, an `msgb` structure and a `datab` structure. It carries data or information, as identified by its message type, in a Stream.

message queue

A linked list of zero or more messages connected together.



message type

A defined set of values identifying the contents of a message.

module

A defined set of kernel-level routines and data structures used to process data, status and control information on a Stream. It is an optional element, but there can be many modules in one Stream. It consists of a pair of queues (read queue and write queue), and it communicates to other components in a Stream by passing messages.

multiplexer

A STREAMS mechanism that allows messages to be routed among multiple Streams in the kernel. A multiplexing configuration includes at least one multiplexing pseudo-device driver connected to one or more upper Streams and one or more lower Streams.

named Stream

A Stream, typically a pipe, with a name associated with it via a call to `fattach(3C)` (that is, a mount operation). This is different from a named pipe (FIFO) in two ways: a named pipe (FIFO) is unidirectional while a named Stream is bidirectional; a named Stream need not refer to a pipe but can be another type of a Stream.

open routine

A procedure in each STREAMS driver and module called by STREAMS on each `open(2)` system call made on the Stream. A module's open procedure is also called when the module is pushed.

packet mode

A feature supported by the STREAMS-based pseudo-terminal subsystem. It is used to inform a process on the master side when state changes occur on the slave side of a pseudo-TTY. It is enabled by pushing a module called `pckt` on the master side.

persistent link

A connection below a multiplexer that can exist without having an open controlling Stream associated with it.

pipe

Same as a STREAMS-based pipe.

pop

A term used when a module that is immediately below the Stream head is removed.

pseudo-device driver

A software driver, not directly associated with a physical device, that performs functions internal to a Stream such as a multiplexer or log driver.

pseudo-terminal subsystem

A user interface identical to a terminal subsystem except that there is a process in a place of a hardware device. It consists of at least a master device, slave device, line discipline module, and hardware emulation module.

push

A term used when a module is inserted in a Stream immediately below the Stream head.

pushable module

A module put between the Stream head and driver. It performs intermediate transformations on messages flowing between the Stream head and driver. A driver is a non-pushable module.

put procedure

A routine in a module or driver associated with a queue which receives messages from the preceding queue. It is the single entry point into a queue from a preceding queue. It may perform processing on the message and will then generally either queue the message for subsequent processing by this queue's `service` procedure, or will pass the message to the `put` procedure of the following queue.

queue

A data structure that contains status information, a pointer to routines processing messages, and pointers for administering a Stream. It typically contains pointers to a `put` and `service` procedure, a message queue, and private data.



read-side

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

read queue

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

remote mode

A feature available with the pseudo-terminal subsystem. It is used for applications that perform the canonical and echoing functions normally done by the line discipline module and tty driver. It enables applications on the master side to turn off the canonical processing.

SAD

A STREAMS Administrative Driver that provides an interface to the autopush mechanism.

schedule

To place a queue on the internal list of queues which will subsequently have their `service` procedure called by the STREAMS scheduler. STREAMS scheduling is independent of the Solaris process scheduling.

service interface

A set of primitives that define a service at the boundary between a service user and a service provider and the rules (typically represented by a state machine) for allowable sequences of the primitives across the boundary. At a Stream/user boundary, the primitives are typically contained in the control part of a message; within a Stream, in `M_PROTO` or `M_PCPROTO` message blocks.

service procedure

A routine in module or driver associated with a queue that receives messages queued for it by the `put` procedure of that queue. The procedure is called by the STREAMS scheduler. It may perform processing on the message and generally passes the message to the `put` procedure of the following queue.



service provider

An entity in a service interface that responds to request primitives from the service user with response and event primitives.

service user

An entity in a service interface that generates request primitives for the service provider and consumes response and event primitives.

slave driver

A STREAMS-based device supported by the pseudo-terminal subsystem. It is also called `pts` and works with a line discipline module and hardware emulation module to provide an interface to a user process.

standard pipe

A mechanism for a unidirectional flow of data between two processes where data written by one process become data read by the other process.

Stream

A kernel aggregate created by connecting STREAMS components, resulting from an application of the STREAMS mechanism. The primary components are the Stream head, the driver, and zero or more pushable modules between the Stream head and driver.

STREAMS-based pipe

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

Stream end

A Stream component furthest from the user process, containing a driver.

Stream head

A Stream component closest to the user process. It provides the interface between the Stream and the user process.



STREAMS

A kernel mechanism that provides the framework for network services and data communication. It defines interface standards for character input/output within the kernel, and between the kernel and user level. The STREAMS mechanism comprises integral functions, utility routines, kernel facilities, and a set of structures.

TTY driver

A STREAMS-based device used in a terminal subsystem.

upper Stream

A Stream that terminates above a multiplexer. The beginning of an upper Stream originates at the Stream head or another multiplexer driver.

upstream

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

watermark

A limit value used in flow control. Each queue has a high watermark and a low watermark. The high watermark value indicates the upper limit related to the number of bytes contained on the queue. When the queued character reaches its high watermark, STREAMS causes another queue that attempts to send a message to this queue to become blocked. When the characters in this queue are reduced to the low watermark value, the other queue will be unblocked by STREAMS.

write queue

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

write-side

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

Index

A

- acknowledgment message, 40
- adjmsg, 350
- alloca, 351
- assembly programming, 7
- asynchronous input/output
 - in polling, 116
- asynchronous protocol Stream
 - example, 46 to ??
- audience intended, 1
- autopush, 34

B

- back-enable of a queue, 78
- back-enabling, 78
- background job
 - in job control, 119
- backq, 351
- bcanput, 351
- bidirectional transfer
 - example, 146 to 151
- bufcall, 352, 368

C

- canonical processing, 47

- canput, 352
- character I/O, 9
- character processing, 47
- cloning (STREAMS), 195
- close
 - dismantling the Stream, 35
- connld(7), 256
- controlling terminal, 123
- copyb, 353
- copymsg, 353
- copyreq structure, 325
- copyresp structure, 326

D

- datamsg, 354
- device
 - close, 9
 - open, 9
- difference between driver & module, 38
- downstream
 - definition, 8
- driver, 6
 - classification, 182
 - ioctl control, 39
 - overview, 181 to 184
 - STREAMS, 184 to ??

- writing a driver, 182 to ??
- driver STREAMS, 18
- dupb, 354
- dupmsg, 354

E

- ECHOCTL, 262
- enableok, 355
- entry point address, 31
- esballoc, 355
- EUC handling in `ldterm(7)`, 265
- expedited data, 55, 156
- extended STREAMS buffers, 105 to ??
 - allocation, 106
 - freeing, 106

F

- `fattach(3C)`, 251
- `fdetach(3C)`, 252
- FIFO (STREAMS), 245
 - basic operations, 246 to 249
 - flush, 250
 - queue scheduling, 45
- file descriptor, 9
- file descriptor passing, 253
- flow control, 76 to 81
 - definition, 11
 - in driver, 194
 - in line discipline module, 176
 - in module, 175 to 176
 - routines, ?? to 81
- flush handling
 - description, 153 to 155
 - flags, 153, 342
 - in driver, 190
 - in line discipline, 155
 - in pipes and FIFOs, 250
 - priority band data, 156
 - read-side example, 154
 - write-side example, 154
- flushband, 356
- flushq, 356

- foreground job
 - in job control, 119
- freeb, 356
- freemsg, 357
- full-duplex processing, 5

G

- `getmsg(2)`, 61
- `getpmsg` function, 63
- `getq`, 357
- `grantpt(3C)`, 281
 - with pseudo-tty driver, 277

H

- hardware emulation module, 268 to 270

I

- `I_SWROPT`, 248
- infinite loop
 - service procedure, 46
- input/output polling, 111 to 117
- `insq`, 358
- ioctl structure, 325
 - with `M_IOCTL`, 331
- `ioctl I_SWROPT`, 248
- `ioctl(2)`
 - `I_POP`, 35
 - `I_PUSH`, 35
 - `TIOCREMOTE`, 276
 - `TIOCSIGNAL`, 280
- `ioctl(2)`
 - general processing, 132 to 133
 - handled by `ptem(7)`, 273
 - hardware emulation module, 269
 - `I_ATMARK`, 70
 - `I_CANPUT`, 70
 - `I_CKBAND`, 69
 - `I_GETBAND`, 69
 - `I_LINK`, 216, 331
 - `I_LIST`, 151
 - `I_PLINK`, 331

- I_PUNLINK, 331
- I_RECVFD, 253, 334
- I_SENDFD, 253, 334
- I_SETSIG events, 117
- I_STR, 40, 331
- I_STR processing, 134 to 135
- I_UNLINK, 221, 331
 - supported by ldterm(7), 264
 - supported by master driver, 279
 - transparent, 135 to 151
- isastream(3C), 253

J

- job control, 119 to 122
 - terminology, 119 to 120

K

- kernel thread, 45

L

- ldterm(7), 261
- LIFO
 - module add/remove, 38
- line discipline module
 - close, 262
 - description, 261 to 268
 - in job control, 121
 - in pseudo-tty subsystem, 271
 - ioctl(2), 264
 - open, 262
- link editing, 7
- linkb, 358
- linkblk structure, 327
- linked list, 30
- lower multiplexer, 20

M

- M_BREAK, 329
- M_COPYIN, 340
 - transparent ioctl example, 139 to 143
- M_COPYOUT, 341
 - transparent ioctl example, 143 to 146
 - with M_IOCTL, 333
- M_CTL, 330
 - with line discipline module, 261
- M_DATA, 14, 330
- M_DELAY, 330
- M_ERROR, 341
- M_FLUSH, 342
 - flags, 342
 - in module example, 173
 - packet mode, 277
- M_HANGUP, 343
- M_IOCACK, 344
 - with M_COPYOUT, 341
 - with M_IOCTL, 332
- M_IOCADATA, 344
- M_IOCNAK, 345
 - with M_COPYOUT, 341
 - with M_IOCTL, 332
- M_IOCTL, 331 to 334
 - transparent, 332
 - with M_COPYOUT, 341
- M_PASSFP, 334
- M_PCPROTO, 14, 346
- M_PCRSE, 346
- M_PCSIG, 346
- M_PROTO, 14, 334 to 335
- M_READ, 346
- M_RSE, 335
- M_SETOPTS
 - SO_READOPT options, 64
- M_SETOPTS, 336 to 339
 - SO_FLAG, 336 to 339
 - SO_WROFF value, 65
 - with ldterm(7), 262
- M_SIG, 339
 - in signaling, 118
- M_START, 347
- M_STARTI, 347
- M_STOP, 347
- M_STOPI, 347
- manipulating modules, 24

master driver
 in pseudo-tty subsystem, 270
 open, 277

message (STREAMS), 13
 allocation, 99
 control information, 14, 94
 definition, 8
 freeing, 100
 handled by `pckt(7)`, 276
 handled by `psem(7)`, 273
 high priority, 54, 340 to 347
 `ldterm(7)` read side, 263
 `ldterm(7)` write side, 264
 linkage, 57
 M_DATA, 14
 M_PCPROTO, 14
 M_PROTO, 14
 ordinary, 54, 329 to 340
 processing, 75
 put back on queue, 46
 recovering from allocation failure, 102
 sending/receiving, 59
 service interface, 82 to 94
 structures, 56, 324 to 325
 types, 13, 53

message block (STREAMS), 8
 linkage, 57

message processing routines (STREAMS), ?? to 46
 design guidelines, 162 to 164

message queue (STREAMS)
 priority, 15, 66 to 70

minor device, 9

module, 16 to 18
 control information, 6
 definition, 6
 difference with driver, 38
 draining, 36
 inserting, 36
 insertion, 7
 ioctl control, 39
 manipulation, 24
 pushed, 8
 reusability, 26
 status information, 6

MORECTL, 94

MOREDATA, 94

msgdsize, 359

multiplexer
 building, 212 to 220
 controlling Stream, 217
 data routing, 222
 declarations, 229
 definition, 19
 design guidelines, 243
 driver, 228 to 238
 example, 225 to 227
 lower, 211
 lower connection, 223 to 224
 lower disconnection, 224
 lower read put procedure, 237 to 238
 lower write service procedure, 236
 upper, 211
 upper write put procedure, 232 to ??
 upper write service procedure, 235

multiplexer ID
 in multiplexer building, 216
 in multiplexer dismantling, 221

multiplexing STREAMS, 18 to 23

N

named pipe (see FIFO), 245

named Stream
 description, ?? to 253
 `fattach(3C)`, 251
 `fdetach(3C)`, 252
 file descriptor passing, 253
 `isastream(3C)`, 253
 remote, 254

noenable, 359

NSTRPUSH, 35

O

O_NDELAY, 36
 with M_SETOPTS, 338

O_NONBLOCK, 36
 with M_SETOPTS, 338

open
 device file, 33
 organization of guide, 1
 other documentation, 4
 OTHERQ, 359

P

packet mode
 description, 276
 messages, 276
 pkt(7), 276
 PIPE_BUF, 249
 pipemod STREAMS module, 250
 pipes, 6
 STREAMS (see STREAMS-based
 pipe), 245
 point-to-point, 9
 pollfd structure, 114
 polling
 error events, 115
 events, 112
 example, 113 to 116
 priority band data, 55, 156
 flush handling example, 156
 ioctl(2), 69
 routines, 68
 processing
 canonical, 47
 character, 47
 read-side, 49
 protocol
 migration, 25
 protocol
 portability, 24
 substitution, 25
 pseudo-device driver, 20
 pseudo-tty emulation module, 272 to 276
 pseudo-tty subsystem, 270
 description, 270 to 282
 drivers, 277 to 280
 ldterm(7), 271
 messages, 273
 packet mode, 276
 remote mode, 276
 ptem structure, 275
 ptem(7), 272, 275
 ptm (see master driver), 270
 pts (see slave driver), 270
 ptsname(3C), 281
 with pseudo-tty driver, 278
 pullupmsg, 359
 put procedure, 13, 44
 putbq, 360
 putctl, 360
 putctl1, 361
 putmsg(2), 60
 putpmsg function, 62
 putq, 362

Q

qenable, 363
 qreply, 363
 qsize, 364
 queue, 8, 12
 data structures, 31
 limits, 33
 pairs, 8

R

RD, 364
 read side
 definition, 8
 ldterm(7) messages, 263
 ldterm(7) processing, 263
 put procedure, 168
 read-side processing, 49
 releasing callback requests, 105
 rmvb, 364
 rmvq, 364

S

SAD (see STREAMS Administrative Driver), 395

scheduler

 STREAMS, 45

service interface, 23, 83 to 85

 definition, 82

 library example, 86 to 94

 rules, 94

service primitive, 85

 in service procedure, 87

service procedure, 13, 45, 80

 infinite loop, 46

service provider, 85

 accessing, 88

 closing, 91

 receiving data, 92

 sending data, 91

signal(2), 111

signals

 extended, 118

 in job control management, 121

 in STREAMS, 118

slave driver

 in pseudo-tty subsystem, 270

 open, 278

SO_FLAG

 in M_SETOPTS, 336 to 339

stdata

 data structure, 34

strapush structure, 395

strbuf structure, 61

strchg(1), 151

strconf command, 151

STRCTLSZ parameter, 394

Stream

 controlling terminal, 122

 hung-up, 122

 simple, 5

Stream construction, 5, 30 to 36

 add/remove modules, 35

 close a Stream, 35

 define module/driver, 33

 example, 36 to 41

 open a Stream, 33

 queue structures, 31

Stream definition, 5

Stream head

 definition, 5

 intercepting I_STR, 40

 processing control, 64

STREAMS, 3

 basic operations, 9 to 12

 configuration, 381 to 397

 manual pages, 399

 mechanisms, 29

 message queues, 14

 system calls, 9, 29

 tunable parameters, 393 to ??

STREAMS Administrative Driver, 395 to 397

STREAMS benefits, 23 to 27

STREAMS components, 12 to 18

STREAMS data structures, 321 to 327

 design, 165

 dynamic allocation, 165

STREAMS debugging, 376 to 379

 error and trace logging, 377 to 379

STREAMS definition, 5

STREAMS driver, 18, 184 to ??

 cloning, 195

 close routine design, 161

 declarations, 126

 definition, 6

 design guidelines, 160 to ??, 209

 environment, 125

 flow control, 194

 flush handling, 190

 ioctl(2), 130 to 152

 loop-around, 198 to ??

 open routine design, 161

 printer driver example, 187 to ??

 pseudo-tty, 277 to 280

 pseudo-tty subsystem master, 270

 pseudo-tty subsystem slave, 270

STREAMS message queues, 14

 priority, 15 to 16

-
- STREAMS module, 16 to 18, 167 to 175
 - autopush facility, 394, 397
 - close routine design, 161
 - connld(7), 256
 - control information, 6
 - declarations, 126
 - definition, 6
 - design guidelines, 160 to ??, 178
 - environment, 125
 - filter, 171
 - flow control, 175 to ??
 - ioctl(2), 130 to 152
 - line discipline, 261
 - null module example, 128
 - open routine design, 161
 - ptem(7), 272
 - read side put procedure, 168
 - routines, 167 to 171
 - service interface example, 94 to 99
 - service procedure, 170
 - status information, 6
 - write side put procedure, 169
 - STREAMS multiplexing, 18 to 23
 - STREAMS queue
 - definition, 8
 - flags, 72, 73
 - overview, 12
 - qband structure, 72
 - queue structure, 70
 - structures, 322 to 324
 - using equeue information, 73
 - using qband information, 73
 - STREAMS scheduler, 45
 - in service procedure, 45
 - STREAMS utility routines, 349 to 367
 - STREAMS-based pipe, 6
 - atomic write, 249
 - basic operations, 246 to 249
 - creation, 34
 - creation errors, 246
 - definition, 245
 - PIPE_BUF, 249
 - STREAMS-based pseudo-terminal subsystem (see pseudo-tty subsystem), 270
 - STREAMS-based terminal subsystem (see tty subsystem), 259
 - striioctl structure, 40, 326
 - strlog, 365
 - STRMSGSZ parameter, 394
 - stroptions structure, 327
 - strqget, 365
 - strqset, 366
 - synchronous input/output
 - in polling, 112
- ## T
- termio(7), 121
 - default flag values, 261
 - testb, 366
 - thread
 - service procedure, 45
 - transparent ioctl
 - M_COPYIN example, 139 to 143
 - M_COPYOUT example, 143 to 146
 - messages, 138
 - processing, 135 to 151
 - tty subsystem
 - benefits, 259
 - description, 259 to 270
 - hardware emulation module, 268 to 270
 - ldterm(7), 261
 - setup, 260
- ## U
- unbufcall, 366, 369
 - unique connection (STREAMS), 254 to 257
 - unlinkb, 367
 - unlockpt(3C), 281
 - with pseudo-tty driver, 278
 - upper multiplexer, 20
 - upper Stream, 19
 - upstream
 - definition, 8

W

WR, 367

write side

definition, 8

ldterm(7), 264

put procedure, 169