

Linux Fast-STREAMS STREAMS Programmer's Guide

Version 0.7a Edition 4
Updated 2006-01-04
Package streams-0.7a.4

Brian Bidulock <bidulock@openss7.org> for
The OpenSS7 Project <<http://www.openss7.org/>>

Copyright © 2001-2005 OpenSS7 Corporation <<http://www.openss7.com/>>
Copyright © 1997-2000 Brian F. G. Bidulock <bidulock@openss7.org>
All Rights Reserved.

Published by OpenSS7 Corporation
1469 Jefferys Crescent
Edmonton, Alberta T6L 6T1
Canada

This is texinfo edition 4 of the Linux Fast-STREAMS documentation, and is consistent with streams 0.7a. This guide was developed under the **OpenSS7 Project** and was funded in part by **OpenSS7 Corporation**.

Permission is granted to make and distribute verbatim copies of this guide provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this guide under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this guide into another language, under the same conditions as for modified versions.

Short Contents

Acknowledgements	1
Preface	3
1 Introduction	11
2 Overview	39
3 Mechanism	41
4 Processing	59
5 Messages	65
6 Polling	93
7 Modules and Drivers	95
8 Modules	97
9 Drivers	99
10 Multiplexing	101
11 Pipes and FIFOs	103
12 Terminal Subsystem	105
13 Synchronization	107
14 Reference	109
15 Conformance	111
16 Portability	113
A Data Structures	115
B Message Types	117
C Utilities	119
D Debugging	121
E Configuration	123
F Administration	125
G Examples	127
H Copying	129
Glossary	143
List of Figures	151
List of Listings	153
Index	155

Table of Contents

Acknowledgements	1
Sponsors	1
Contributors	1
Preface	3
Document Information	3
Notice	3
Abstract	3
Objective	3
Intent	3
Audience	3
Revisions	4
Version Control	4
ISO 9000 Compliance	5
Disclaimer	5
U.S. Government Restricted Rights	5
Organization	6
Conventions Used	8
Other Documentation	8
UNIX Edition	9
Related Manuals	9
Copyright	9
1 Introduction	11
1.1 Background	11
1.2 What is STREAMS?	12
1.2.1 Characteristics	12
1.2.2 Components	13
1.2.2.1 Stream head	14
1.2.2.2 Module	15
1.2.2.3 Driver	16
1.2.2.4 Queues	16
1.2.2.5 Messages	16
1.3 Basic Streams Operations	17
1.3.1 Basic Operations Example	18
1.4 Components	20
1.4.1 Queues	20
1.4.1.1 Queue Procedures	21
1.4.2 Messages	22
1.4.2.1 Message Types	22
1.4.2.2 Message Linkage	22
1.4.2.3 Message Queueing Priority	24

1.4.3	Modules	25
1.4.4	Drivers	27
1.4.5	Stream Head	28
1.5	Multiplexing	29
1.5.1	Fan-Out Multiplexers	30
1.5.2	Fan-In Multiplexers	31
1.5.3	Complex Multiplexers	32
1.6	Benefits of STREAMS	32
1.6.1	Standardized Service Interfaces	33
1.6.2	Manipulating Modules	34
1.6.2.1	Protocol Portability	34
1.6.2.2	Protocol Substitution	35
1.6.2.3	Protocol Migration	36
1.6.2.4	Module Reusability	37
2	Overview	39
2.1	Definitions	39
2.2	Concepts	39
2.3	Application Interface	39
2.4	Kernel Level Facilities	39
2.5	Subsystems	39
3	Mechanism	41
3.1	Mechanism Overview	41
3.1.1	STREAMS System Calls	42
3.2	Stream Construction	42
3.2.1	Opening a STREAMS Device File	45
3.2.1.1	First Open of a Stream	46
3.2.1.2	Subsequent Open of a Stream	47
3.2.2	Opening a STREAMS-based FIFO	48
3.2.3	Creating a STREAMS-based Pipe	49
3.2.4	Adding and Removing Modules	50
3.2.4.1	Pushing Modules	50
3.2.4.2	Popping Modules	51
3.2.5	Closing the Stream	51
3.2.6	Stream Construction Example	52
3.2.6.1	Inserting Modules	52
3.2.6.2	Module and Driver Control	54
3.2.6.3	Stream Dismantling with Modules	56
3.2.6.4	Stream Construction Example Summary	57
4	Processing	59
4.1	Procedures	59
4.1.1	Put Procedure	59
4.1.2	Service Procedure	61
4.1.3	Put and Service Procedure Summary	62
4.2	Asynchronous Example	63

5	Messages	65
5.1	Messages Overview	65
5.1.1	Message Types	65
5.1.1.1	Ordinary Messages	66
5.1.1.2	High Priority Messages	66
5.1.2	Expedited Data	67
5.2	Message Structure	67
5.2.1	Message Linkage	71
5.2.2	Sending and Receiving Messages	73
5.2.2.1	putmsg(2)	74
5.2.2.2	getmsg(2)	75
5.2.2.3	putpmsg(2s)	76
5.2.2.4	getpmsg(2s)	76
5.2.3	Control of Stream Head Processing	77
5.2.3.1	Read Options	78
5.2.3.2	Read Mode	78
5.2.3.3	Read Protocol	78
5.2.3.4	Write Options	79
5.2.3.5	Write Offset	80
5.3	Queues and Priority	80
5.3.1	Queue Priority Utilities	81
5.3.1.1	strqget(9)	82
5.3.1.2	strqset(9)	83
5.3.2	Queue Priority Commands	84
5.3.2.1	I_FLUSHBAND	85
5.3.2.2	I_CKBAND	85
5.3.2.3	I_GETBAND	85
5.3.2.4	I_CANPUT	85
5.3.2.5	I_ATMARK	86
5.3.2.6	I_GETSIG	88
5.3.2.7	I_SETSIG	88
5.3.3	The queue Structure	88
5.3.3.1	Using queue Information	89
5.3.3.2	queue Flags	89
5.3.4	The qband Structure	90
5.3.4.1	Using qband Information	91
5.3.5	Message Processing	91
5.3.5.1	Flow Control	91
5.3.6	Scheduling	91
5.3.6.1	Flow Control Variables	91
5.3.6.2	Flow Control Procedures	91
5.3.6.3	The STREAMS Scheduler	91
5.4	Service Interfaces	91
5.4.1	Service Interface Benefits	91
5.4.2	Service Interface Library Example	91
5.4.2.1	Accessing the Service Provider	92
5.4.2.2	Closing the Service Provider	92
5.4.2.3	Sending Data to the Service Provider	92

5.4.2.4	Receiving Data	92
5.4.2.5	Module Service Interface Example	92
5.5	Message Allocation	92
5.5.1	Recovering From No Buffers	92
5.6	Extended Buffers	92
6	Polling	93
6.1	Input and Output Polling	93
6.2	Controlling Terminal	93
7	Modules and Drivers	95
7.1	Environment	95
7.2	Input-Output Control	95
7.3	Flush Handling	95
7.4	Driver-Kernel Interface	95
7.5	Design Guidelines	95
8	Modules	97
8.1	Module	97
8.2	Module Flow Control	97
8.3	Module Design Guidelines	97
9	Drivers	99
9.1	External Device Numbers	99
9.2	Internal Device Numbers	99
9.3	spec File System	99
9.4	Clone Device	99
9.5	Named STREAMS Device	99
9.6	Driver	99
9.7	Cloning	99
9.8	Loop-Around Driver	99
9.9	Driver Design Guidelines	99
10	Multiplexing	101
10.1	Multiplexors	101
10.2	Connecting and Disconnecting Lower Stream	101
10.3	Multiplexor Construction Example	101
10.4	Multiplexing Driver	101
10.5	Persistent Links	101
10.6	Multiplexing Driver Design Guidelines	101
11	Pipes and FIFOs	103
11.1	Pipes and FIFOs	103
11.2	Flushing Pipes and FIFOs	103
11.3	Named Streams	103
11.4	Unique Connections	103

12	Terminal Subsystem	105
12.1	Terminal Subsystem	105
12.2	Pseudo-Terminal Subsystem	105
13	Synchronization	107
13.1	MT Configuration	107
13.2	Asynchronous Entry Points	107
13.3	Asynchronous Callbacks	107
13.4	Synchronous Entry Points	107
13.5	Synchronous Callbacks	107
13.6	STREAMS Framework Integrity	107
13.7	MP Message Ordering	107
13.8	MP-UNSAFE Modules	107
13.9	MP Put and Service Procedures	107
13.10	MP Timeout and Buffer Callbacks	107
13.11	MP Open and Close Procedures	107
13.12	MP Module Unloading	107
13.13	MP Locking	107
13.14	MP Asynchronous Callbacks	107
13.15	Stream Integrity	107
14	Reference	109
14.1	Files	109
14.2	System Modules	109
14.3	System Drivers	109
14.4	System Calls	109
14.5	Input-Output Controls	109
14.6	Module Entry Points	109
14.7	Structures	109
14.8	Registration	109
14.9	Message Handling	109
14.10	Queue Handling	109
14.11	Miscellaneous Functions	109
14.12	Extensions	109
14.13	Compatibility	109
15	Conformance	111
15.1	SVR 4.2 Compatibility	111
15.2	AIX Compatibility	111
15.3	HP-UX Compatibility	111
15.4	OSF/1 Compatibility	111
15.5	UnixWare Compatibility	111
15.6	Solaris Compatibility	111
15.7	SUX Compatibility	111
15.8	UXP Compatibility	111
15.9	LiS Compatibility	111

16	Portability	113
16.1	Core Function Support	113
16.2	SVR 4.2 Portability	113
16.3	AIX Portability	113
16.4	HP-UX Portability	113
16.5	OSF/1 Portability	113
16.6	UnixWare Portability	113
16.7	Solaris Portability	113
16.8	SUX Portability	113
16.9	UXP Portability	113
16.10	LiS Portability	113
Appendix A	Data Structures	115
A.1	Stream Structures	115
A.2	Queue Structures	115
A.3	Message Structures	115
A.4	Input Output Control Structures	115
A.5	Link Structures	115
A.6	Options Structures	115
Appendix B	Message Types	117
B.1	Message Type	117
B.2	Ordinary Messages	117
B.3	High Priority Messages	117
Appendix C	Utilities	119
Appendix D	Debugging	121
Appendix E	Configuration	123
Appendix F	Administration	125
F.1	Administrative Utilities	125
F.2	System Controls	125
F.3	/proc File System	125
Appendix G	Examples	127
G.1	Module Example	127
G.2	Driver Example	127

Appendix H	Copying	129
H.1	GNU General Public License	129
H.1.1	Preamble	129
H.1.2	Terms and Conditions for Copying, Distribution and Modification	130
H.1.3	How to Apply These Terms to Your New Programs	134
H.2	GNU Free Documentation License	135
H.2.1	Preamble	135
H.2.2	Terms and Conditions for Copying, Distribution and Modification	135
H.2.3	How to use this License for your documents	141
Glossary		143
List of Figures		151
List of Listings		153
Index		155

Acknowledgements

As with most open source projects, this project would not have been possible without the valiant efforts and productive software for the *Free Software Foundation* and the *Linux Kernel Community*.

Sponsors

Funding for completion of the Linux Fast-STREAMS package was provided in part by:

- OpenSS7 Corporation

Additional funding for **The OpenSS7 Project** was provided by:

- OpenSS7 Corporation
- Lockheed Martin
- Performance Technologies
- Motorola
- HOB International
- Comverse
- Sonus Networks
- France Telecom
- SS8 Networks
- Nortel Networks
- Verisign

Contributors

The primary contributor to the OpenSS7 Linux Fast-STREAMS package is **Brian F. G. Bidulock**. The following is a list of significant contributors to **The OpenSS7 Project**:

- Per Berquist
- John Boyd
- Chuck Winters
- Peter Courtney
- Tom Chandler
- Gurol Ackman
- Kutluk Testicioglu
- Others

Acknowledgements

Preface

Document Information

Notice

This package is released and distributed under the *GNU General Public License* (see [Section H.1 \[GNU General Public License\], page 129](#)). Please note, however, that there are different licensing terms for the manual pages and some of the documentation (derived from OpenGroup¹ publications and other sources). Consult the permission notices contained in the documentation for more information.

This document, is released under the *GNU Free Documentation License* (see [Section H.2 \[GNU Free Documentation License\], page 135](#)) with all sections invariant.

Abstract

This document provides a *STREAMS Programmer's Guide* for *Linux Fast-STREAMS*.

Objective

The objective of this document is to provide a guide for the *STREAMS* programmer when developing *STREAMS* modules, drivers and application programs for *Linux Fast-STREAMS*.

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

STREAMS was incorporated in UNIX System V Release 3 to augment the character input/output (I/O) mechanism and to support development of communication services.

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

Intent

The intent of this document is to act as an introductory guide to the *STREAMS* programmer. It is intended to be read alone and is not intended to replace or supplement the *Linux Fast-STREAMS* manual pages. For a reference for writing code, the manual pages (see *STREAMS(9)*) provide a better reference to the programmer. Although this describes the features of the *Linux Fast-STREAMS* package, **OpenSS7 Corporation** is under no obligation to provide any software, system or feature listed herein.

Audience

This document is intended for a highly technical audience. The reader should already be familiar with *Linux* kernel programming, the *Linux* file system, character devices, driver input and output, interrupts, software interrupt handling, scheduling, process contexts, multiprocessor locks, etc.

¹ Formerly X/Open and UNIX International.

Preface

The guide is intended for network and systems programmers, who use the *STREAMS* mechanism at user and kernel levels for *Linux* and *UNIX* system communication services. Readers of the guide are expected to possess prior knowledge of the *Linux* and *UNIX* system, programming, networking, and data communication.

Revisions

Take care that you are working with a current version of this document: you will not be notified of updates. To ensure that you are working with a current version, contact the [Author](#), or check [The OpenSS7 Project](#) website for a current version.

A current version of this document is normally distributed with the *Linux Fast-STREAMS* package.

Version Control

```
SPG2.texi,v
Revision 0.9.2.6  2005/11/20 22:20:18  brian
- still working up documentation

Revision 0.9.2.5  2005/11/17 10:52:33  brian
- working up take 2

Revision 0.9.2.4  2005/11/17 01:59:26  brian
- more workup of take 2

Revision 0.9.2.3  2005/11/16 10:30:39  brian
- still working up take 2

Revision 0.9.2.2  2005/11/16 03:20:03  brian
- working up take 2

Revision 0.9.2.1  2005/11/15 12:05:09  brian
- second run at SPG

Revision 0.9.2.45 2005/11/14 23:27:06  brian
- more workup

Revision 0.9.2.44 2005/11/14 11:19:49  brian
- working up manual

Revision 0.9.2.43 2005/11/14 04:43:55  brian
- updating manual

Revision 0.9.2.42 2005/11/13 23:04:01  brian
- starting cleanup of SPG

Revision 0.9.2.41 2005/10/07 09:34:00  brian
- more testing and corrections

Revision 0.9.2.40 2005/09/26 10:56:41  brian
- doc updates

Revision 0.9.2.39 2005/09/20 12:53:07  brian
```



```
- more doc updates, corrected QFULL handling

Revision 0.9.2.38  2005/09/18 07:38:35  brian
- more doc updates

Revision 0.9.2.37  2005/09/17 11:52:08  brian
- documentation updates

Revision 0.9.2.36  2005/09/17 08:20:57  brian
- more doc updates

Revision 0.9.2.35  2005/09/17 00:46:12  brian
- document updates

Revision 0.9.2.34  2005/09/16 03:06:02  brian
- added glossary

Revision 0.9.2.33  2005/09/15 13:02:52  brian
- added new graphics and updates
```

ISO 9000 Compliance

Only the `TeX`, `texinfo`, or `roff` source for this document is controlled. An opaque (printed, postscript or portable document format) version of this document is an **UNCONTROLLED VERSION**.

Disclaimer

OpenSS7 Corporation disclaims all warranties with regard to this documentation including all implied warranties of merchantability, fitness for a particular purpose, non-infringement, or title; that the contents of the document are suitable for any purpose, or that the implementation of such contents will not infringe on any third party patents, copyrights, trademarks or other rights. In no event shall *OpenSS7 Corporation* be liable for any direct, indirect, special or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with any use of this document or the performance or implementation of the contents thereof.

OpenSS7 Corporation reserves the right to revise this software and documentation for any reason, including but not limited to, conformity with standards promulgated by various agencies, utilization of advances in the state of the technical arts, or the reflection of changes in the design of any techniques, or procedures embodied, described, or referred to herein. *OpenSS7 Corporation* is under no obligation to provide any feature listed herein.

U.S. Government Restricted Rights

If you are licensing this Software on behalf of the U.S. Government ("Government"), the following provisions apply to you. If the Software is supplied by the Department of Defense ("DoD"), it is classified as "Commercial Computer Software" under paragraph 252.227-7014 of the DoD Supplement to the Federal Acquisition Regulations ("DFARS") (or any successor regulations) and the Government is acquiring only the license rights granted herein (the

license rights customarily provided to non-Government users). If the Software is supplied to any unit or agency of the Government other than DoD, it is classified as "Restricted Computer Software" and the Government's rights in the Software are defined in paragraph 52.227-19 of the Federal Acquisition Regulations ("FAR") (or any successor regulations) or, in the cases of NASA, in paragraph 18.52.227-86 of the NASA Supplement to the FAR (or any successor regulations).

Organization

This guide has several chapters, each discussing a unique topic. [Chapter 1 \[Introduction\]](#), page 11, [Chapter 2 \[Overview\]](#), page 39, [Chapter 3 \[Mechanism\]](#), page 41 and [Chapter 4 \[Processing\]](#), page 59 contain introductory information and can be ignored by those already familiar with *STREAMS* concepts and facilities.

This document is organized as follows:

[\[Preface\]](#), page 3

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 1 \[Introduction\]](#), page 11

An introduction to *STREAMS* and the *Linux Fast-STREAMS* package. *STREAMS* Fundamentals. Presents an overview and the benefits of *STREAMS*.

[Chapter 2 \[Overview\]](#), page 39

A brief overview of *STREAMS*.

[Chapter 3 \[Mechanism\]](#), page 41

A description of the *STREAMS* framework. 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 \[Processing\]](#), page 59

Processing and procedures within the *STREAMS* framework. Gives an overview of the *STREAMS* put and service routines.

[Chapter 5 \[Messages\]](#), page 65

STREAMS Messages, organization, types, priority, queueing, and general handling. Discusses *STREAMS* messages, their structure, linkage, queueing, and interfacing with other *STREAMS* components.

[Chapter 6 \[Polling\]](#), page 93

Polling of *STREAMS* file descriptors and other asynchronous application techniques. Describes how *STREAMS* allows user processes to monitor, control, and poll Streams to allow an effective utilization of system resources.

[Chapter 7 \[Modules and Drivers\]](#), page 95

An overview of *STREAMS* modules, drivers and multiplexing drivers. Describes the *STREAMS* module and driver environment, input-output controls,

routines, declarations, flush handling, driver-kernel interface, and also provides general design guidelines for modules and drivers.

Chapter 8 [Modules], page 97

Details of *STREAMS* modules, including examples. Provides information on module construction and function.

Chapter 9 [Drivers], page 99

Details of *STREAMS* drivers, including examples. Discusses *STREAMS* drivers, elements of driver flow control, flush handling, cloning, and processing.

Chapter 10 [Multiplexing], page 101

Details of *STREAMS* multiplexing drivers, including examples. Describes the *STREAMS* multiplexing facility.

Chapter 11 [Pipes and FIFOs], page 103

Details of *STREAMS*-based Pipes and FIFOs. Provides information on creating, writing, reading, and closing of *STREAMS*-based pipes and FIFOs and unique connections.

Chapter 12 [Terminal Subsystem], page 105

Details of *STREAMS*-based Terminals and Pseudo-terminals. Discusses *STREAMS*-based terminal and pseudo-terminal subsystems.

Chapter 13 [Synchronization], page 107

Discusses *STREAMS* in a symmetrical multi-processor environment.

Chapter 14 [Reference], page 109

Reference section.

Chapter 15 [Conformance], page 111

Conformance of the *Linux Fast-STREAMS* package to other *UNIX* implementations of *STREAMS*.

Chapter 16 [Portability], page 113

Portability of *STREAMS* modules and drivers written for other *UNIX* implementations of *STREAMS* and how they can most easily be ported into *Linux Fast-STREAMS*; but, for more details on this topic, see the *Linux Fast-STREAMS - STREAMS Portability Guide*.

Appendix A [Data Structures], page 115

Primary *STREAMS* Data Structures, descriptions of their members, flags, constants and use. Summarizes data structures commonly used by *STREAMS* modules and drivers.

Appendix B [Message Types], page 117

STREAMS Message Type reference, with descriptions of each message type. Describes *STREAMS* messages and their use.

Appendix C [Utilities], page 119

STREAMS kernel-level utility functions for the module or driver writer. Describes *STREAMS* utility routines and their usage.

Appendix D [Debugging], page 121

STREAMS debugging facilities and their use. Provides debugging aids for developers.

Appendix E [Configuration], page 123

STREAMS configuration, the *STREAMS Administrative Driver* and the autopush facility. Describes how modules and drivers are configured into the *Linux* and *UNIX* system, tunable parameters, and the autopush facility.

Appendix F [Administration], page 125

Administration of the *STREAMS* subsystem.

Appendix G [Examples], page 127

Collected examples.

Conventions Used

This guide uses *texinfo* typographical conventions.

Throughout this guide, the word *STREAMS* will refer to the mechanism and the word *Stream* will refer 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, often reference fictional drivers and modules. Some examples are also present in the *Linux Fast-STREAMS* package, both for testing and example purposes.

System calls, *STREAMS* utility routines, header files, and data structures are given using *texinfo* ‘filename’ typesetting, when they are mentioned in the text.

Variable names, pointers, and parameters are given using *texinfo* *variable* typesetting conventions. Routine, field, and structure names unique to the examples are also given using *texinfo* *variable* typesetting conventions when they are mentioned in the text.

Declarations and short examples are in *texinfo* ‘sample’ typesetting.

texinfo displays are used to show program source code.

Data structure formats are also shown in *texinfo* displays.

Other Documentation

Although the *STREAMS Programmer’s Guide for Linux Fast-STREAMS* provides a guide to aid in developing *STREAMS* applications, readers are encouraged to consult the *Linux Fast-STREAMS* manual pages. For a reference for writing code, the manual pages (see *STREAMS(9)*) provide a better reference to the programmer. For detailed information on system calls used by *STREAMS* (section 2), and *STREAMS* utilities from section 8. *STREAMS* specific input output control (ioctl) calls are provided in *streamio(7)*. *STREAMS* modules and drivers are described on section 7. *STREAMS* is also described to some extent in the *System V Interface Definition, Third Edition*.

UNIX Edition

This system conforms to *UNIX System V Release 4.2* for *Linux*.

Related Manuals

Linux Fast-STREAMS Installation and Reference Manual

Copyright

© 1997-2005 OpenSS7 Corporation. All Rights Reserved.

1 Introduction

1.1 Background

STREAMS is a facility first presented in a paper by Dennis M. Ritchie in 1984,¹ originally implemented on 4.1BSD and later part of *Bell Laboratories Eighth Edition UNIX*, incorporated into *UNIX System V Release 3.0* and enhanced in *UNIX System V Release 4* and *UNIX System V Release 4.2*. *STREAMS* was used in *SVR4* for terminal input/output, pseudo-terminals, pipes, named pipes (FIFOs), interprocess communication and networking. Since its release in *System V Release 4*, *STREAMS* has been implemented across a wide range of *UNIX*, *UNIX*-like, and *UNIX*-based systems, making its implementation and use an *ipso facto* standard.

STREAMS is a facility that allows for a reconfigurable full duplex communications path, *Stream*, between a user process and a driver in the kernel. Kernel protocol modules can be pushed onto and popped from the *Stream* between the user process and driver. The *Stream* can be reconfigured in this way by a user process. The user process, neighbouring protocol modules and the driver communicate with each other using a message passing scheme closely related to *MOM* (*Message Oriented Middleware*). This permits a loose coupling between protocol modules, drivers and user processes, allowing a third-party and loadable kernel module approach to be taken toward the provisioning of protocol modules on platforms supporting *STREAMS*.

On *UNIX System V Release 4.2*, *STREAMS* was used for terminal input-output, pipes, FIFOs (named pipes), and network communications. Modern *UNIX*, *UNIX*-like and *UNIX*-based systems providing *STREAMS* normally support some degree of network communications using *STREAMS*; however, many do not support *STREAMS*-based pipe and FIFOs² or terminal input-output.³

Linux has not traditionally implemented a *STREAMS* subsystem. It is not clear why, however, perceived ideological differences between *STREAMS* and *Sockets* and also the *XTI/TLI* and *Sockets* interfaces to *Internet Protocol* services are usually at the centre of the debate. For additional details on the debate, see [section “About This Manual” in *Linux Fast-STREAMS Frequently Asked Questions*](#).

Linux pipes and FIFOs are *SVR3*-style, and the *Linux* terminal subsystem is *BSD*-like. *UNIX 98 Pseudo-Terminals*, ‘*ptys*’, have a specialized implementation that does not follow the *STREAMS* framework and, therefore, do not support the pushing or popping of *STREAMS* modules. Internal networking implementation under *Linux* follows the *BSD* approach with a native (system call) *Sockets* interface only.

RedHat at one time provided an *Intel Binary Compatibility Suite (iBCS)* module for *Linux* that supported the *XTI/TLI* interface and ‘*socksys*’ system calls and input-output controls, but not the *STREAMS* framework (and therefore cannot push or pop modules).

¹ *A Stream Input-Output System*, AT&T Bell Laboratories Technical Journal 63, No. 8 Part 2 (October, 1984), pp. 1897-1910.

² For example, AIX.

³ For example, HP-UX

A *STREAMS* package for *Linux* was written and eventually distributed and maintained by **GCOM Inc.** This is the *Linux STREAMS (LiS)* package. This package had some failings and was repeatedly rejected for mainline adoption, which prompted the development of *Linux Fast-STREAMS*. *Linux STREAM (LiS)* is no longer supported (it does not have a maintainer).

Linux Fast-STREAMS is the current open source implementation of *STREAMS* for *Linux* and provides all of the capabilities of *UNIX System V Release 4.2 MP*, plus support for mainstream *UNIX* implementations based on *UNIX System V Release 4.2 MP* through compatibility modules.

Although it is intended primarily as documentation for the *Linux Fast-STREAMS* implementation of *STREAMS*, much of the *Linux Fast-STREAMS - STREAMS Programmer's Guide* is generally applicable to all *STREAMS* implementations.

1.2 What is STREAMS?

STREAMS is a flexible, message oriented framework for the development of *GNU/Linux* communications facilities and protocols. It provide a set of system calls, kernel resources, and kernel utilities within a framework that is applicable to a wide range of communications facilities including terminal subsystems, interprocess communication, and networking. It provides standard interfaces for communication input and output within the kernel, common facilities for device drivers, and a standard interface⁴ between the kernel and the rest of the *GNU/Linux* system.

The standard interface and mechanism enable modular, portable development and easy integration of high performance network services and their components. Because it is a message passing architecture, *STREAMS* does not impose a specific network architecture (as does the *BSD Sockets* kernel architecture. The *STREAMS* user interface is uses the familiar *UNIX* character special file input and output mechanisms `open(2)`, `read(2)`, `write(2)`, `ioctl(2)`, `close(2)`; and provides additional system calls, `poll(2)`, `getmsg(2)`, `getpmsg(2s)`, `putmsg(2)`, `putpmsg(2s)`, to assist in message passing between user-level applications and kernel-resident modules. Also, *STREAMS* defines a standard set of input-output controls (`ioctl(2)`) for manipulation and configuration of *STREAMS* by a user-space application.

As a message passing architecture, the *STREAMS* interface between the user process and kernel resident modules can be treated either as fully synchronous exchanges or can be treated asynchronously for maximum performance.

1.2.1 Characteristics

STREAMS has the the following characteristics that are not exhibited (or are exhibited in different ways) by other kernel level subsystems:

- *STREAMS* is based on the character device special file which is one of the most flexible special files available in the *GNU/Linux* system.

⁴ XPG 4.2/XNS 4.2, XPG 5/XNS 5, POSIX/SUSv2 XSI Extensions and POSIX/SUSv3 XSR Extensions.

- *STREAMS* is a message passing architecture, similar to *Message Oriented Middleware (MOM)* that achieves a high degree of functional decoupling between modules. This allows the service interface between modules to correspond to the natural interfaces found or described between protocol layers in protocol stack without requiring the implementation to conform to any given model.

As a contrasting example, the *BSD Sockets* implementation, internal to the kernel, provides strict socket-protocol, protocol-protocol and protocol-device function call interfaces.

- By using `put` and `service` procedures for each module, and scheduling `service` procedures, *STREAMS* combines background scheduling of coroutine service procedures with message queueing and flow control to provide a mechanism robust for both event driven subsystem and soft real-time subsystem.

In contrast, *BSD Sockets*, internal to the kernel, requires the sending component across the socket-protocol, protocol-protocol, or protocol-device to handle flow control. *STREAMS* integrates flow control within the *STREAMS* framework.

- *STREAMS* permits user runtime configuration of kernel data structure and modules to provide for a wide range of novel configurations and capabilities in a live *GNU/Linux* system. The *BSD Sockets* protocol framework does not provide this capability.
- *STREAMS* is as applicable to termination input-output and interprocess communication as it is to networking protocols.

BSD Sockets is only applicable to a restricted range of networking protocols.

- *STREAMS* provides mechanisms (the pushing and popping of modules, and the linking and unlinking of *Streams* under multiplexing drivers) for complex configuration of protocol stacks; the precise topology being typically under the control of user space daemon processes.

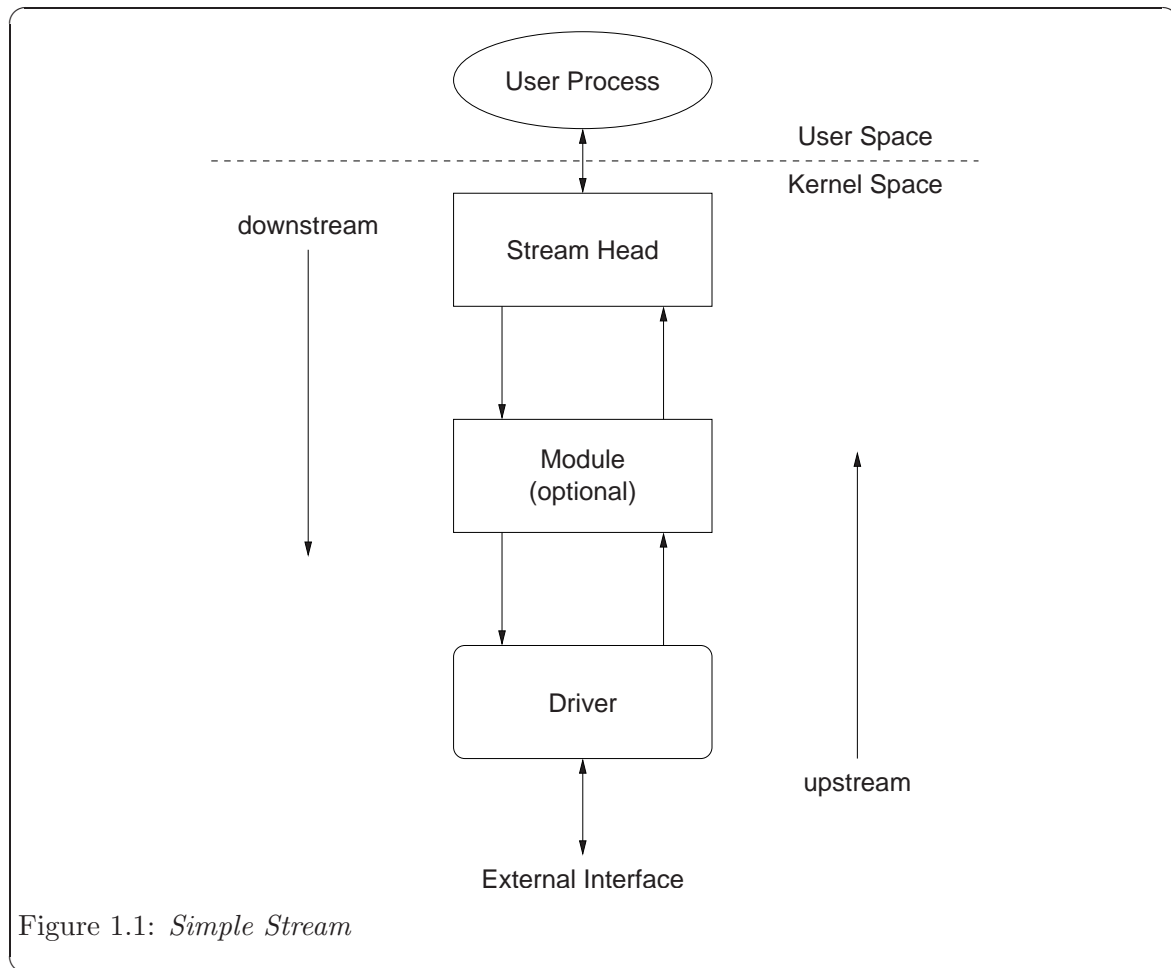
No other kernel protocol stack framework provides this flexible capability. Under *BSD Sockets* it is necessary to define specialized socket types to perform these configuration functions and not in any standard way.

1.2.2 Components

STREAMS provides a full-duplex communications path for data and control information between a kernel-resident driver and a user space process (see [Figure 1.1](#)).

Within the kernel, a *Stream* is comprised of the following basic components:

- A *Stream head* that is inside the *Linux* kernel, but which sits closest to the user space process. The *Stream head* is responsible for communicating with user space processes and that presents the standard *STREAMS* I/O interface to user space processes and applications.
- A *Stream end* or *Driver* that is inside the *Linux* kernel, but which sits farthest from the user space process. A *Stream end* or *Driver* that interfaces to hardware or other mechanisms within the *Linux* kernel.
- A *Module* that sits between the *Stream head* and *Stream end*. The *Module* provides modular and flexible processing of control and data information passed up and down the *Stream*.



1.2.2.1 Stream head

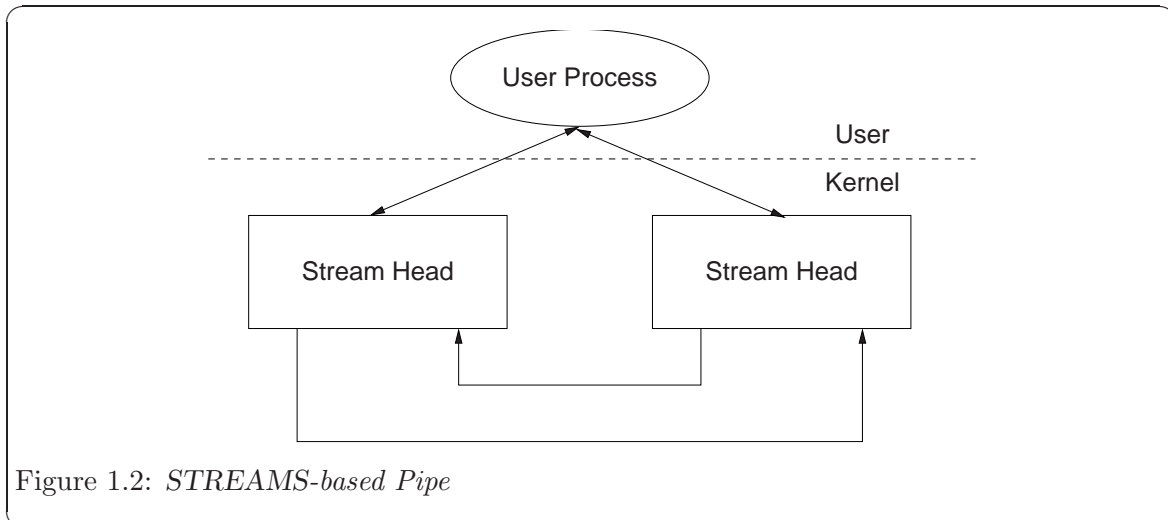
A *Stream head* is the component of a *Stream* that is closest to the user space process. The *Stream head* is responsible for directly communicating with the user space process in user context and for converting system calls to actions performed on the *Stream head* or the conversion of control and data information passed between the user space process and the *Stream* in response to system calls. All *Streams* are associated with a *Stream head*. In the case of *STREAMS*-based pipes, the *Stream* may be associated with two (interconnected) *Stream heads*. Because the *Stream head* follows the same structure as a *Module*, it can be viewed as a specialized module.

With *STREAMS*, pipes and FIFOs are also *STREAMS*-based.⁵ *STREAMS*-based pipes and FIFOs do not have a *Driver* component.

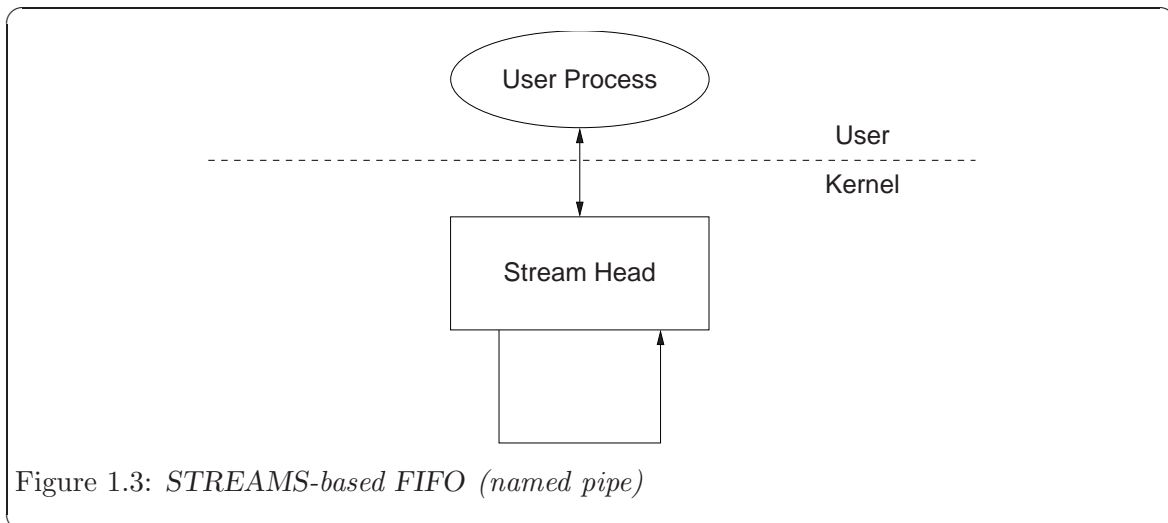
STREAMS-based pipes place another *Stream head* in the position of the *Driver*. That is, a *STREAMS*-based pipe is a full-duplex communications path between two otherwise independent *Stream heads*. *Modules* may be placed between the *Stream heads* in the same

⁵ Unlike the native *Linux* pipes and FIFOs that use the older *UNIX System V Release 3* or *BSD* approaches to these facilities.

fashion as they can exist between a *Stream head* and a *Driver* in a normal *Stream*. A *STREAMS*-based pipe is illustrated in [Figure 1.2](#).



STREAMS-based FIFOs consist of a single *Stream head* that has its downstream path connected to its upstream path where the *Driver* would be located. *Modules* can be pushed under this single *Stream Head*. A *STREAMS*-based FIFO is illustrated in [Figure 1.3](#).



For more information on *STREAMS*-based pipes and FIFOs, see [Chapter 11 \[Pipes and FIFOs\]](#), page 103.

1.2.2.2 Module

A *STREAMS* *Module* is an optional processing element that is placed between the *Stream head* and the *Stream end*. The *Module* can perform processing functions on the data and control information flowing in either direction on the *Stream*. It can communicate with neighbouring modules, the *Stream head* or a *Driver* using *STREAMS* messages. Each

Module is self-contained in the sense that it does not directly invoke functions provided by, nor access datastructures of, neighbouring modules, but rather communicates data, status and control information using messages. This functional isolation provides a loose coupling that permits flexible recombination and reuse of *Modules*. A *Module* follows the same framework as the *Stream head* and *Driver*, has all of the same entry points and can use all of the same *STREAMS* and kernel utilities to perform its function.

Modules can be inserted between a *Stream head* and *Stream end* (or another *Stream head* in the case of a *STREAMS*-based pipe or FIFO). The insertion and deletion of *Modules* from a *Stream* is referred to as *pushing* and *popping* a *Module* due to the fact that that modules are inserted or removed from just beneath the *Stream head* in a push-down stack fashion. Pushing and popping of modules can be performed using standard `ioctl(2)` calls and can be performed by user space applications without any need for kernel programming, assembly, or relinking.

For more information on *STREAMS Modules*, see [Section 1.4.3 \[Modules\]](#), page 25.

1.2.2.3 Driver

All *Streams*, with the sole exception of *STREAMS*-based pipe and FIFOs, contain a *Driver* at the *Stream end*. A *STREAMS Driver* can either be a *device driver* that directly or indirectly controls hardware, or can be a *pseudo-device driver* that interface with other software subsystems within the kernel. *STREAMS* drivers normally perform little processing within the *STREAMS* framework and typically only provide conversion between *STREAMS* messages and hardware or software events (e.g. interrupts) and conversion between *STREAMS* framework data structures and device related data structures.

For more information on *STREAMS Drivers*, see [Section 1.4.4 \[Drivers\]](#), page 27.

1.2.2.4 Queues

Each component in a *Stream* (*Stream head*, *Module*, *Driver*) has an associated pair of queues. One *queue* in each pair is responsible for managing the message flow in the *downstream* direction from *Stream head* to *Stream end*; the other for the *upstream* direction. The *downstream queue* is called the *write-side queue* in the *queue* pair; the *upstream queue*, the *read-side queue*.

Each *queue* in the pair provides pointers necessary for organizing the temporary storage and management of *STREAMS* messages on the *queue*, as well as function pointers to procedures to be invoked when messages are placed on the *queue* or need to be taken off of the *queue*, and pointers to auxillary and module-private data structures. The *read-side queue* also contains function pointers to procedures used to `open` and `close` the *Stream head*, *Module* or *Driver* instance associated with the *queue* pair. *Queue* pairs are dynamically allocated when an instance of the *driver*, *module* or *Stream head* is created and deallocated when the instance is destroyed.

For more information on *STREAMS Queues*, see [Section 1.4.1 \[Queues\]](#), page 20.

1.2.2.5 Messages

STREAMS is a message passing architecture. *STREAMS* messages can contain control information or data, or both. Messages that contain control information are intended to

illicit a response from a neighbouring module, *Stream head* or *Stream end*. The control information typically uses the message type to invoke a general function and the fields in the control part of the message as arguments to a call to the function. The data portion of a message represents information that is (from the perspective of the *STREAMS* framework) unstructured. Only cooperating modules, the *Stream head* or *Stream end* need know or agree upon the format of control or data messages.

A *STREAMS* message consists of one or more blocks. Each block is a 3-tuple of a message block, a data block and a data buffer. Each data block has a message type, and the data buffer contains the control information or data associated with each block in the message. *STREAMS* messages typically consist of one control-type block (`M_PROTO`) and zero or more data-type blocks (`M_DATA`), or just a data-type block.

A set of specialized and standard message types define messages that can be sent by a *module* or *driver* to control the *Stream head*. A set of specialized and standard message types define messages that can be sent by the *Stream head* to control a *module* or *driver*, normally in response to a standard input-output control for the *Stream*.

STREAMS messages are passed between a module, *Stream head* or *Driver* using a `put` procedure associated with the queue in the queue pair for the direction in which the message is being passed. Messages passed towards the *Stream head* are passed in the *upstream* direction, and those towards the *Stream end*, in the *downstream* direction. The *read-side* queue in the queue pair associated with the module instance to which a message is passed is responsible for processing or queueing *upstream* messages; the *write-side* queue, for processing *downstream* messages.

STREAMS messages are generated by the *Stream head* and passed *downstream* in response to `write(2)`, `putmsg(2)`, and `putpmsg(2s)` system calls; they are also consumed by the *Stream head* and converted to information passed to user space in response to `read(2)`, `getmsg(2)`, and `getpmsg(2s)` system calls.

STREAMS messages are also generated by the *Driver* and passed *upstream* to ultimately be read by the *Stream head*; they are also consumed when written by the *Stream head* and ultimately arrive at the *Driver*.

For more information on *STREAMS Messages*, see [Section 1.4.2 \[Messages\]](#), page 22.

1.3 Basic Streams Operations

This section provides a basic description of the user level interface and system calls that are used to manipulate a *Stream*.

A *Stream* is similar, and indeed is implemented, as a character device special file and is associated with a character device within the *GNU/Linux* system. Each *STREAMS* character device special file (character device node, see `mknod(2)`) has associated with it a major and minor device number. In the usual situation, a *Stream* is associated with each minor character device node in a similar fashion to a minor device instance for regular character device drivers.

STREAMS devices are opened, as are character device drivers, with the `open(2)` system call.⁶ Opening a minor device node accesses a separate *Stream* instance between the user level process and the *STREAMS* device driver. As with normal character devices, the file descriptor returned from the `open(2)` call, can be used to further access the *Stream*.

Opening a minor device node for the first time results in the creation of a new instance of a *Stream* between the *Stream head* and the *driver*. Subsequent opens of the same minor device node does not result in the creation of a new *Stream*, but provides another file descriptor that can be used to access the same *Stream* instance. Only the first open of a minor device node will result in the creation of a new *Stream* instance.

Once it has opened a *Stream*, the user level process can send and receive data to and from the *Stream* with the usual `read(2)` and `write(2)` system calls that are compatible with the existing character device interpretations of these system calls. *STREAMS* also provides the additional system calls, `getmsg(2)` and `getpmsg(2s)`, to read control and data information from the *Stream*, as well as `putmsg(2)` and `putpmsg(2s)` to write control and data information. These additional system calls provide a richer interface to the *Stream* than is provided by the traditional `read(2)` and `write(2)` calls.

A *Stream* is closed using the `close(2)` system call (or a call that closes file descriptors such as `exit(2)`). If a number of processes have the *Stream* open, only the last `close(2)` of a *Stream* will result in the destruction of the *Stream* instance.

1.3.1 Basic Operations Example

An basic example of opening, reading from and writing to a *Stream* driver is shown in [Listing 1.1](#).

⁶ An exception is *STREAMS*-based pipes, that are opened with the `pipe(2)` system call.

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

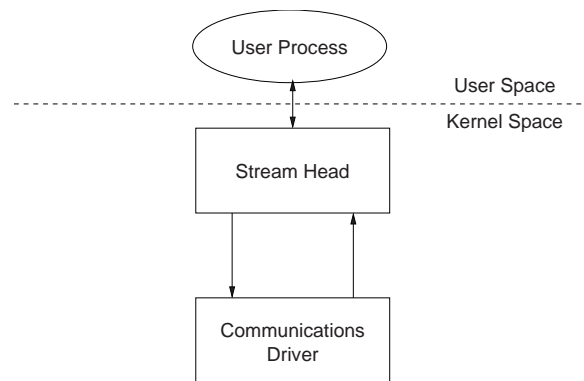
    if ((fd = open("/dev/streams/comm/1", O_RDWR)) < 0) {
        perror("open failed");
        exit(1);
    }

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

Listing 1.1: *Basic Operations*

The example in [Listing 1.1](#) is for a communications device that provide a communications channel for data transfer between two processes or hosts. Data written to the device is communicated over the channel to the remote process or host. Data read from the device was written by the remote process or host.

In the example in [Listing 1.1](#), a simple *Stream* is opened using the `open(2)` call. `‘/dev/streams/comm/1’` is the path to the character minor device node in the file system. When the device is opened, the character device node is recognized as a *STREAMS* special file, and the *STREAMS* subsystem creates a *Stream* (if one does not already exist for the minor device node) and associates it with the minor device node. [Figure 1.4](#) illustrates the state of the *Stream* at the point after the `open(2)` call returns.

Figure 1.4: *Stream to Communications Driver*

The `while` loop in [Listing 1.1](#) simply reads data from the device using the `read(2)` system call and then writes the data back to the device using the `write(2)` system call.

When a *Stream* is opened for blocking operation (i.e., neither `O_NONBLOCK` nor `O_NDELAY` were set), `read(2)` will block until some data arrives. The `read(2)` call might, however, return less than the requested ‘1024’ bytes. When data is read, the routine simply writes the data back to the device.

STREAMS implements flow control both in the upstream and downstream directions. Flow control limits the amount of normal data that can be queued awaiting processing within the *Stream*. High and low water marks for flow control are set on a queue pair basis. Flow control is local and specific to a given *Stream*. High priority control messages are not subject to *STREAMS* flow control.

When a *Stream* is opened for blocking operation (i.e., neither `O_NONBLOCK` nor `O_NDELAY` were set), `write(2)` will block while waiting for flow control to subside. `write(2)` will always block awaiting the availability of *STREAMS* message blocks to satisfy the call, regardless of the setting of `O_NONBLOCK` or `O_NDELAY`.

In the example in [Listing 1.1](#), the `exit(2)` system call is used to exit the program; however, the `exit(2)` results in the equivalent of a call to `close(2)` for all open file descriptors and the *Stream* is flushed and destroyed before the program is finally exited.

1.4 Components

This section briefly describes each *STREAMS* component and how they interact within a *Stream*. Chapters later in this manual describe the components and their interaction in greater detail.

1.4.1 Queues

This subsection provides a brief overview of message *queues* and their associated procedures.

A *queue* provides an interface between an instance of a *STREAMS* driver, module or *Stream head*, and the other modules and drivers that make up a *Stream* for a direction of message flow (i.e., *upstream* or *downstream*). When an instance of a *STREAMS* driver, module or *Stream head* is associated with a *Stream*, a pair of queues are allocated to represent

the driver, module or *Stream head* within the *Stream*. Queue data structures are always allocated in pairs. The first queue in the pair is the *read-side* or *upstream* queue in the pair; the second queue, the *write-side* or *downstream* queue.

Queues are described in greater detail in [Section 5.3 \[Queues and Priority\]](#), page 80.

1.4.1.1 Queue Procedures

This subsection provides a brief overview of *queue* procedures.

The *STREAMS* module, driver or *Stream head* provides five procedures that are associated with each queue in a queue pair: the **put**, **service**, **open**, **close** and **admin** procedures. Normally the **open** and **close** procedures (and possibly the optional **admin** procedure) are only associated with the *read-side* of the queue pair.

Each queue in the pair has a pointer to a **put** procedure. The **put** procedure is used by *STREAMS* to present a new message to an upstream or downstream queue. At the ends of the *Stream*, the *Stream head* write-side, or *Stream end* read-side, queue **put** procedure is normally invoked using the **put(9)** utility. A module within the *Stream* typically has its **put** procedure invoked by an adjacent module, driver or *Stream head* that uses the **putnext(9)** utility from its own **put** or **service** procedure to pass message to adjacent modules. The **put** procedure of the queue receiving the message is invoked. The **put** procedure decides whether to process the message immediately, queue the message on the message queue for later processing by the queue's **service** procedure, or whether to pass the message to a subsequent queue using **putnext(9)**.

Each queue in the pair has a pointer to an optional **service** procedure. The purpose of a **service** procedure process messages that were deferred by the **put** procedure by being placed on the message queue with utilities such as **putq(9)**. A **service** procedure typically loops through taking messages off of the queue and processing them. The procedure normally terminates the loop when it can not process the current message (in which case it places the message back on the queue with **putbq(9)**), or when there is no longer any messages left on the queue to process. A **service** procedure is optional in the sense that if the **put** procedure never places any messages on the queue, a **service** procedure is unnecessary.

Each queue in the pair also has a pointer to a **open** and **close** procedure; however, the *qi-qopen* and *qi-qclose* pointers are only significant in the *read-side* queue of the queue pair.

The queue **open** procedure for a driver is called each time that a driver (or *Stream head*) is opened, including the first open that creates a *Stream* and upon each successive open of the *Stream*. The queue **open** procedure for a module is called when the module is first pushed onto (inserted into) a *Stream*, and for each successive open of a *Stream* upon which the module has already been pushed (inserted).

The queue **close** procedure for a module is called whenever the module is popped (removed) from a *Stream*. Modules are automatically popped from a *Stream* on the last close of the *Stream*. The queue **close** procedure for a driver is called with the last close of the *Stream* or when the last reference to the *Stream* is relinquished. If the *Stream* is linked under a multiplexing driver (**I_LINK(7)**), or has been named with **fattach(3)**, then the *Stream* will

not be dismantled on the last close and the `close` procedure not called until the *Stream* is eventually unlinked (`I_UNLINK(7)`) or detached (`fdetach(3)`).

Procedures are described in greater detail in [Section 4.1 \[Procedures\]](#), page 59.

1.4.2 Messages

This subsection provides a brief overview of *STREAMS* messages.

In fitting with the concept of function decoupling, all control and data information is passed between *STREAMS* modules, drivers and the *Stream head* using messages. Utilities are provided to the *STREAMS* module writer for passing messages using queue and message pointers. *STREAMS* messages consist of a 3-tuple of a message block structure (`msgb(9)`), a data block structure (`atab(9)`) and a data buffer. The message block structure is used to provide an instance of a reference to a data block and pointers into the data buffer. The data block structure is used to provide information about the data buffer, such as message type, separate from the data contained in the buffer. Messages are normally passed between *STREAMS* modules, drivers and the *Stream head* using utilities that invoke the target module's `put` procedure, such as `put(9)`, `putnext(9)`, `qreply(9)`. Messages travel along a *Stream* with successive invocations of each driver, module and *Stream head*'s `put` procedure.

Messages are described in greater detail in [Section 5.1 \[Messages Overview\]](#), page 65 and [Chapter 5 \[Messages\]](#), page 65.

1.4.2.1 Message Types

This subsection provides a brief overview of *STREAMS* message types.

Each data block (`atab(9)`) is assigned a message type. The message type discriminates the use of the message by drivers, modules and the *Stream head*. Most of the message types may be assigned by a module or driver when it generates a message, and the message type can be modified as a part of message processing. The *Stream head* uses a wider set of message types to perform its function of converting the functional interface to the user process into the messaging interface used by *STREAMS* modules and drivers.

Most of the defined message types (see [Section 5.1.1 \[Message Types\]](#), page 65, and [Appendix B \[Message Types\]](#), page 117) are solely for use within the *STREAMS* framework. A more limited set of message types (`M_PROTO`, `M_PCPROTO` and `M_DATA`) can be used to pass control and data information to and from the user process via the *Stream head*. These message type can be generated and consumed using the `read(2)`, `write(2)`, `getmsg(2)`, `getpmsg(2s)`, `putmsg(2)`, `putpmsg(2s)` system calls and some `streamio(7)` *STREAMS* `ioctl(2)`.

Message types are described in detail in [Section 5.1.1 \[Message Types\]](#), page 65 and [Appendix B \[Message Types\]](#), page 117.

1.4.2.2 Message Linkage

Messages blocks of differing types can be linked together into composite messages as illustrated in [Figure 1.5](#).



Figure 1.5: A Message

Messages, once allocated, or when removed from a queue, exist standalone (i.e., they are not attached to any queue). Messages normally exist standalone when they have been first allocated by an interrupt service routine, or by the *Stream head*. They are placed into the *Stream* by the driver or *Stream head* at the *Stream end* by calling `put(9)`. After being inserted into a *Stream*, message normally only exist standalone in a given queue's `put` or `service` procedures. A queue's `put` or `service` procedure normally do one of the following:

- pass the message along to an adjacent queue with `putnext(9)` or `qreply(9)`;
- process and consume the message by deallocating it with `freemsg(9)`;
- place the message on the queue from the `put` procedure with `putq(9)` or from the `service` procedure using `putbq(9)`.

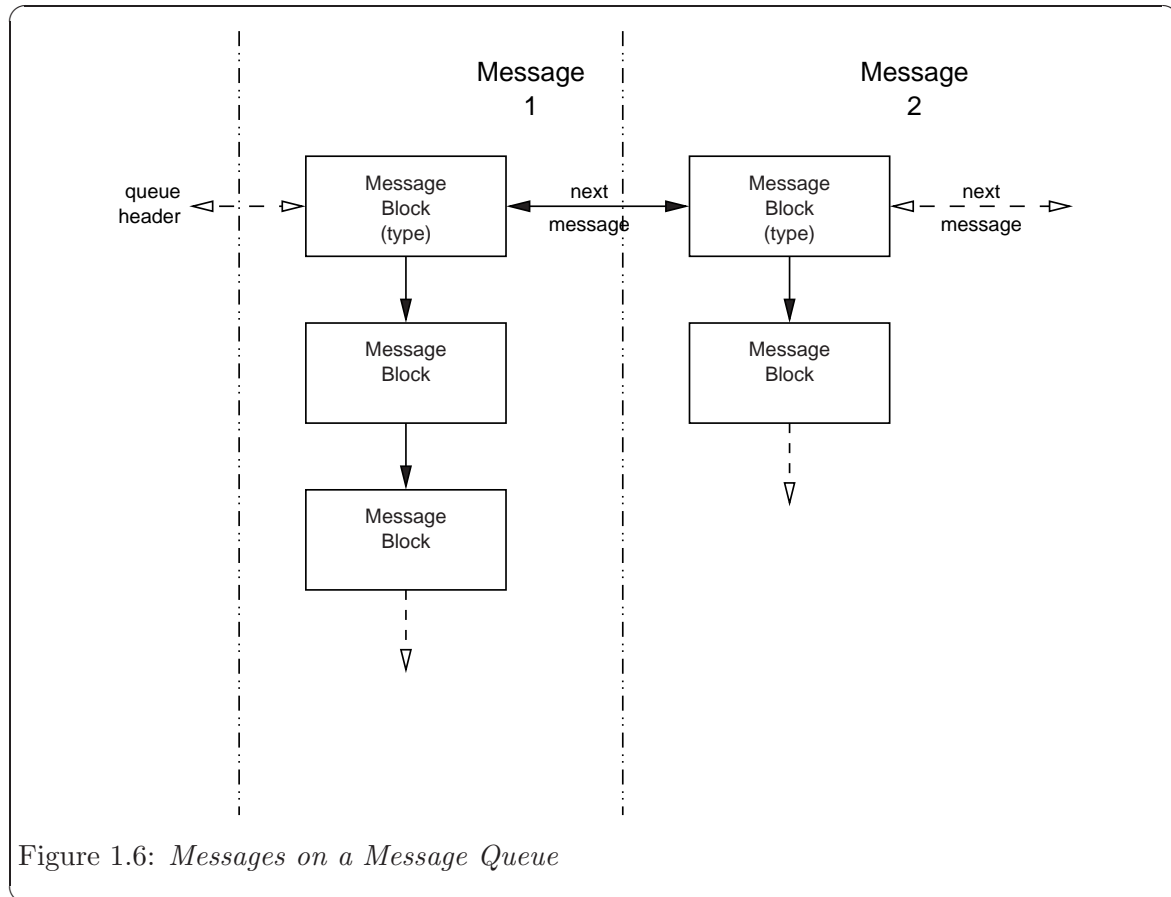
Once placed on a queue, a message exists only on that queue and all other references to the message are dropped.

Only one reference to a message block (`msgb(9)`) exists within the *STREAMS* framework. Additional references to the same data block (`datab(9)`) and data buffer can be established by duplicating the messages block, `msgb(9)` (without duplicating either the data block, `datab(9)`, or data buffer). The *STREAMS* `dupb(9)` and `dupmsg(9)` utilities can be used to duplicate message blocks. Also, the entire 3-tuple of message block, data block and data buffer can be copied using the `copyb(9)` and `copymsg(9)` *STREAMS* utilities.

When a message is first allocated, it is the responsibility of the allocating procedure to either pass the message to a queue `put` procedure, place the message on its own message queue, or free the message. When a message is removed from a message queue, the reference then becomes the responsibility of the procedure that removed it from the queue. Under special circumstances, it might be necessary to temporarily store a reference to a standalone message in a module private data structure, however, this is usually not necessary.

When a message has been placed on a queue, it is linked into the list of messages already on the queue. Messages that exist on a message queue await processing by the queue's `service` procedure. Essentially, queue `put` procedures are a way of performing immediate message processing, and placing a message on a message queue for later processing by the queue's `service` procedure is a way of deferring message processing until a later time: that is, until *STREAMS* schedules the `service` procedure for execution.

Two messages linked together on a message queue is illustrated in [Figure 1.6](#). In the figure, 'Message 2' is linked to 'Message 1'.

Figure 1.6: *Messages on a Message Queue*

As illustrated in [Figure 1.6](#), when a message exists on a message queue, the first message block in the message (which can possibly contain a chain of message blocks) is linked into a double linked list used by the message queue to order and track messages. The queue structure, `queue(9)`, contains the head and tail pointers for the linked list of messages that reside on the queue. Some of the fields in the first message block (such as the linked list pointers) are significant only in the first message block of the message and applies to all the message blocks in the message (such as message band).

Message linkage is described in detail in [Section 5.2 \[Message Structure\]](#), page 67.

1.4.2.3 Message Queueing Priority

This subsection provides a brief overview of *message queueing priority*.

STREAMS message queues provide the ability to process messages of differing priority. There are three classes of message priority (in order of increasing priority):

1. Normal messages.
2. Priority messages.
3. High-priority messages.

Normal messages are queued in priority band '0'. Priority messages are queued in bands greater than zero ('1' through '255' inclusive). Messages of a higher ordinal band number are of greater priority. For example, a priority message for band '23' is queued ahead of

messages for band ‘22’. Normal and priority messages are subject to flow control within a *Stream*, and are queued according to priority.

High priority messages are assigned a priority band of ‘0’; however, their message type distinguishes them as high priority messages and they are queued ahead of all other messages. (The priority band for high priority messages is ignored and always set to ‘0’ whenever a high priority message type is queued.) High priority messages are given special treatment within the *Stream* and are not subjected to flow control; however, only one high priority message can be outstanding for a given transaction or operation within a *Stream*. The *Stream head* will discard high priority messages that arrive before a previous high priority message has been acted upon.

Because queue **service** procedures process messages in the order in which they appear in the queue, messages that are queued toward the head of the queue yield a higher scheduling priority than those toward the tail. High priority messages are queue first, followed by priority messages of descending band numbers, finally followed by normal (band ‘0’) messages.

STREAMS provides independent flow control parameters for ordinary messages. Normal message flow control parameters are contained in the queue structure itself (**queue(9)**); priority parameters, in the auxiliary queue band structure (**qband(9)**). A set of flow control parameters exists for each band (from ‘0’ to ‘255’).

As a high priority message is defined by message type, some message types are available in high-priority/ordinary pairs (e.g., **M_PCPROTO/M_PROTO**) that perform the same function but which have differing priority.

Queueing priority is described in greater detail in [Section 5.3 \[Queues and Priority\]](#), page 80.

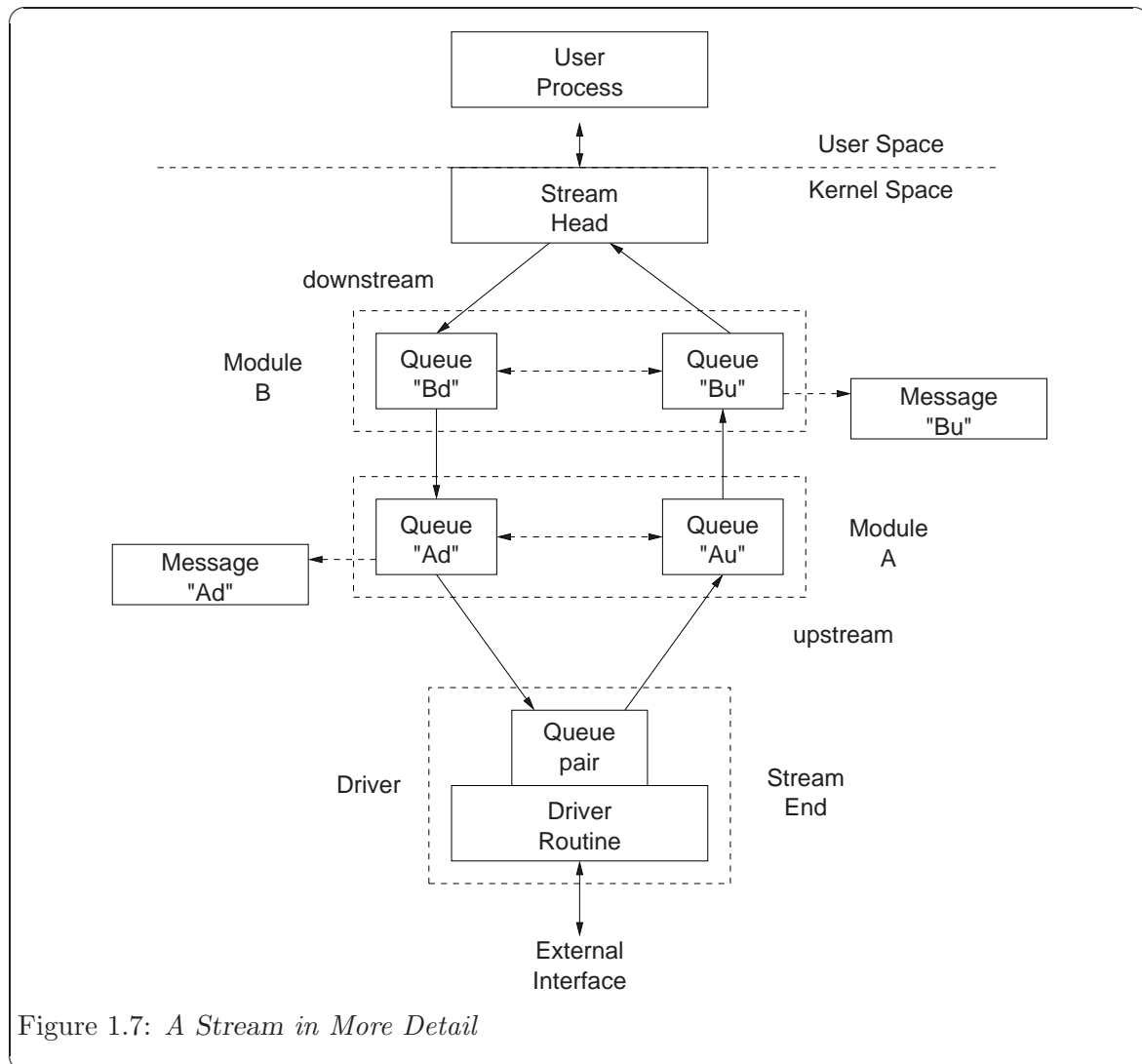
1.4.3 Modules

This subsection provides a brief overview of *STREAMS modules*.

Modules are components of message processing that exist as a unit within a *Stream* beneath the *Stream head*. *Modules* are optional components and zero or more (up to a predefined limit) instances of a module can exist within a given *Stream*. Instances of a *module* have a unique queue pair associated with them that permit the instance to be linked among the other queue pairs in a *Stream*.

[Figure 1.7](#) illustrates and instance each of two modules (‘A’ and ‘B’) that are linked within the same *Stream*. Each module instance consists of a queue pair (‘Ad/Au’ and ‘Bd/Bu’ in the figure). Messages flow from the driver to the *Stream head* through the *upstream* queues in each queue pair (‘Au’ and then ‘Bu’ in the figure); and from *Stream head* to driver through *downstream* queues (‘Bd’ and then ‘Ad’).

The *module* provides unique message processing procedures (**put** and optionally **service** procedures) for each queue in the queue pair. One set of **put** and **service** procedures handles *upstream* messages; the other set, *downstream* messages. Each procedure is independent of the others. *STREAMS* handles the passing of messages but any other information that is to be passed between procedures must be performed explicitly by the procedures themselves. Each queue provides a module private pointer that can be used by procedures for maintaining state information or passing other information between procedures.



Each procedure can pass messages directly to the adjacent queue in either direction of message flow. This is normally performed with the *STREAMS* `putnext(9)` utility. For example, in [Figure 1.7](#), procedures associated with queue ‘Bd’ can pass messages to queue ‘Ad’: ‘Bu’ to ‘Au’.

Also, procedures can easily locate the other queue in a queue pair and pass messages along the opposite direction of flow. This is normally performed using the *STREAMS* `qreply(9)` utility. For example, in [Figure 1.7](#), procedures associated with queue ‘Ad’ can easily locate queue ‘Au’ and pass messages to ‘Bu’ using `qreply(9)`.

Each queue in a module is associated with messages, processing procedures, and module private data. Typically, each queue in the module has a distinct set of message, processing procedures and module private data.

Messages

Messages can be inserted into, and removed from, the linked list message queue associated with each queue in the queue pair as they pass through the module.

For example, in [Figure 1.7](#), ‘Message Ad’ exists on the ‘Ad’ queue; ‘Message Bu’, on the ‘Bu’ queue.

Processing Procedures

Each queue in a *module* queue pair requires that a **put** procedure be defined for the queue. Upstream or downstream modules, drivers or the *Stream head* invoke a **put** procedure of the module when they pass messages to the module along the *Stream*.

Each queue may optionally provide a **service** procedure that will be invoked when messages are placed on the queue for later processing by the **service** procedure. A **service** procedure is never required if the module **put** procedure never enqueues a message to either queue in the queue pair.

Either procedure in either queue in the pair can pass messages upstream or downstream and may alter information within the module private data associated with either queue in the pair.

Data

Module processing procedures can make use of a pointer in each queue structure that is reserved for use by the module writer to locate module private data structures. These data structures are typically attached to each queue from the module’s **open** procedure, and detached from then module’s **close** procedure. Module private data is useful for maintaining state information associated with the instance of the module and for passing information between procedures.

Modules are described in greater detail in [Chapter 8 \[Modules\]](#), page 97.

1.4.4 Drivers

This subsection provides a brief overview of *STREAMS drivers*.

The *Device* component of the *Stream* is an initial part of the regular *Stream* (positioned just below the *Stream head*). Most *Streams* start out life as a *Stream head* connected to a *driver*. The driver is positioned within the *Stream* at the *Stream end*. Note that not all *Streams* require the presence of a driver: a *STREAMS*-based pipe or FIFO *Stream* do not contain a driver component.

A *driver* instance represented by a queue pair within the *Stream*, just as for modules. Also, each queue in the queue pair has a message queue, processing procedures, and private data associated with it in the same way as for *STREAMS* modules. There are three differences that distinguish drivers from modules:

1. Drivers are responsible for generating and consuming messages at the *Stream end*.
Drivers convert *STREAMS* messages into appropriate software or hardware actions, events and data transfer. As a result, drivers that are associated with a hardware device normally contain an interrupt service procedure that handles the external device specific actions, events and data transfer. Messages are typically consumed at the *Stream end* in the driver’s downstream **put** or **service** procedure and action take or data transferred to the hardware device. Messages are typically generated at the

Stream end in the driver's interrupt service procedure, and inserted upstream using the `put(9) STREAMS` utility.

Software drivers (so-called *pseudo-device drivers*) are similar to a hardware device driver with the exception that they typically do not contain an interrupt service routine. Pseudo-device drivers are still responsible for consuming messages at the *Stream end* and converting them into actions and data output (external to *STREAMS*), as well as generating messages in response to events and data input (external to *STREAMS*).

In contrast, *modules* are intended to operate solely within the *STREAMS* framework.

2. Because a driver sits at a *Stream end* and can support multiplexing, a driver can have multiple *Streams* connected to it, either upstream (fan-in) or downstream (fan-out) (see [Section 1.5 \[Multiplexing\]](#), page 29).

In contrast, an instance of a *module* is only connected within a single *Stream* and does not support multiplexing at the module queue pair.

3. An instance of a driver (queue pair) is created and destroyed using the `open(2)` and `close(2)` system calls.

In contrast, an instance of a *module* (queue pair) is created and destroyed using the `I_PUSH` and `I_POP STREAMS ioctl(2)` commands.

Aside from these differences, the *STREAMS driver* is similar in most respects to the *STREAMS module*. Both drivers and modules can pass signals, error codes, return values, and other information to processes in adjacent queue pairs using *STREAMS* messages of various message types provided for that purpose.

Drivers are described in greater detail in [Chapter 9 \[Drivers\]](#), page 99.

1.4.5 Stream Head

This subsection provide a brief overview of *Stream heads*.

The *Stream head* is the first component of a *Stream* that is allocated when a *Stream* is created. All *Streams* have an associated *Stream head*.

In the case of *STREAMS*-based pipes, two *Stream heads* are associated with each other. *STREAMS*-based FIFOs have one *Stream head* but no *Stream end* or *Driver*. For all other *Streams*, as illustrated in [Figure 1.7](#), there exists a *Stream head* and a *Stream end* or *Driver*.

The *Stream head* has a queue pair associated with them, just as does any other *STREAMS* module or driver. Also, just as any other module, the *Stream head* provides the processing procedures and private data for processing of messages passed to queues in the pair.

The differences is that the processing procedures are provided by the *GNU/Linux* system rather than being written by the *module* or *driver* writer. These system provided processing procedures perform the necessary functions to convert generate to and consume messages from the *Stream* in response to system calls invoked by a user process. Also, a set of specialized behaviours are provided and a set of specialized message types that may be exchanged with modules and drivers in the *Stream* to provide the standard interface expected by the user application.

Stream heads are described in greater detail in [Chapter 3 \[Mechanism\]](#), page 41, [Chapter 6 \[Polling\]](#), page 93, [Chapter 11 \[Pipes and FIFOs\]](#), page 103, and [Chapter 12 \[Terminal Subsystem\]](#), page 105.

1.5 Multiplexing

This subsection provides a brief overview of *Stream Multiplexing*.

Basic *Streams* that can be created with the `open(2)` or `pipe(2)` system calls are linear arrangements from *Stream head* to *Driver* or *Stream head* to *Stream head*. Although these linear arrangements satisfy the needs of a large class of *STREAMS* applications, there exists a class of application that are more naturally represented by multiplexing: that is, an arrangements where one or more upper *Streams* feed into one or more lower *Streams*. Network protocol stacks (a significant application are for *STREAMS*) are typically more easily represented by multiplexed arrangements.

A *fan-in* multiplexing arrangement is one in which multiple upper *Streams* feed into a single lower *Stream* in a *many-to-one* relationship as illustrated in [Figure 1.8](#).

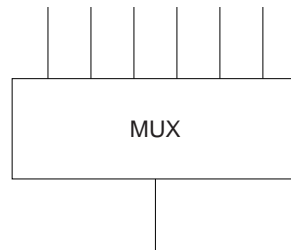


Figure 1.8: *Many-to-one Multiplexor*

A *fan-out* multiplexing arrangement is one in which a single upper *Stream* feeds into multiple lower *Streams* in a *one-to-many* relationship as illustrated in [Figure 1.9](#). (This is the more typically arrangement for communications protocol stacks.)

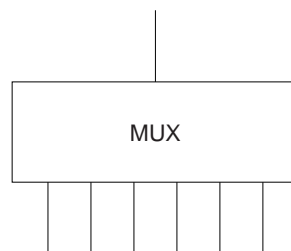
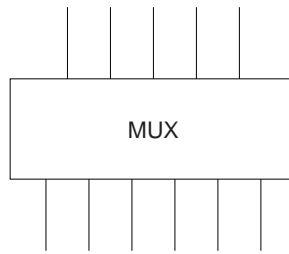


Figure 1.9: *One-to-many Multiplexor*

A *fan-in/fan-out* multiplexing arrangement is one in which multiple upper *Streams* feed into multiple lower *Streams* in a *many-to-many* relationship as illustrated in [Figure 1.10](#).

Figure 1.10: *Many-to-many Multiplexor*

To support these arrangements, *STREAMS* provide a mechanism that can be used to assemble multiplexing arrangements in a flexible way. An, otherwise normal, *STREAMS* pseudo-device driver can be specified to be a multiplexing driver.

Conceptually, a multiplexing driver can perform *upper multiplexing* between multiple *Streams* on its *upper* side connecting the user process and the multiplexing driver, and *lower multiplexing* between multiple *Streams* on its *lower* side connecting the multiplexing driver and the device driver.

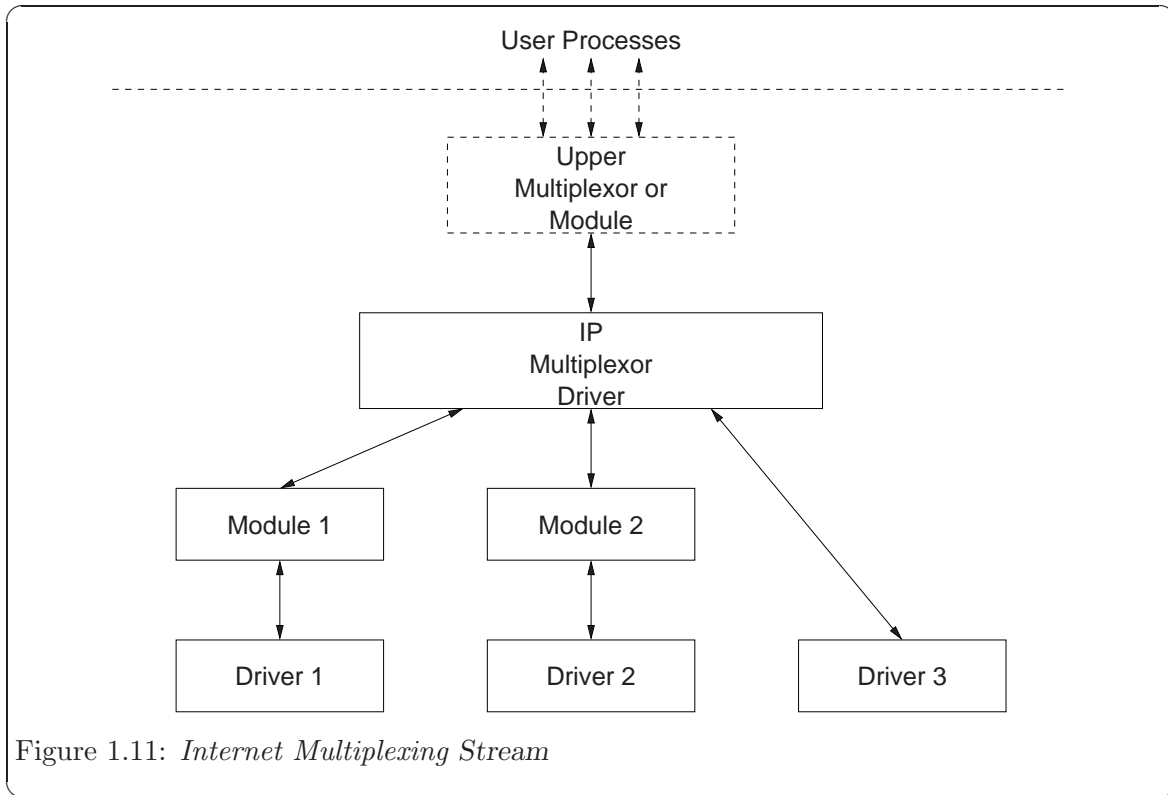
As with normal *STREAMS* drivers, *multiplexing drivers* can have multiple *Streams* created on its *upper* side using the `open(2)` system call. Unlike regular *STREAMS* drivers, however, *multiplexing drivers* have the additional capability that other *Streams* can be linked to the *lower* side of the driver. The linkage is performed by issuing specialized `streamio(7)` commands to the driver that are recognized by multiplexing drivers (`I_LINK`, `I_PLINK`, `I_UNLINK`, `I_PUNLINK`).

Any *Stream* can be linked under a multiplexing driver (provided that it is not already linked under another multiplexing driver). This includes an upper *Stream* of a multiplexing driver. In this fashion, complex trees of multiplexing drivers and linear *Stream* segments containing pushed *modules* can be assembled. Using these linkage commands, complex arrangements can be assembled, manipulated and dismantled by a user or daemon process to suit application needs.

The *fan-in* arrangement of Figure 1.8 performs *upper multiplexing*; the *fan-out* arrangement of Figure 1.9, *lower multiplexing*; and the *fan-in/fan-out* arrangement of Figure 1.10, both *upper* and *lower multiplexing*.

1.5.1 Fan-Out Multiplexers

Figure 1.11 illustrates an example, closely related to the *fan-out* arrangement of Figure 1.9, where the *Internet Protocol (IP)* within a networking stack is implemented as a multiplexing driver and independent *Streams* to three specific device drivers are linked beneath the *IP* multiplexing driver.

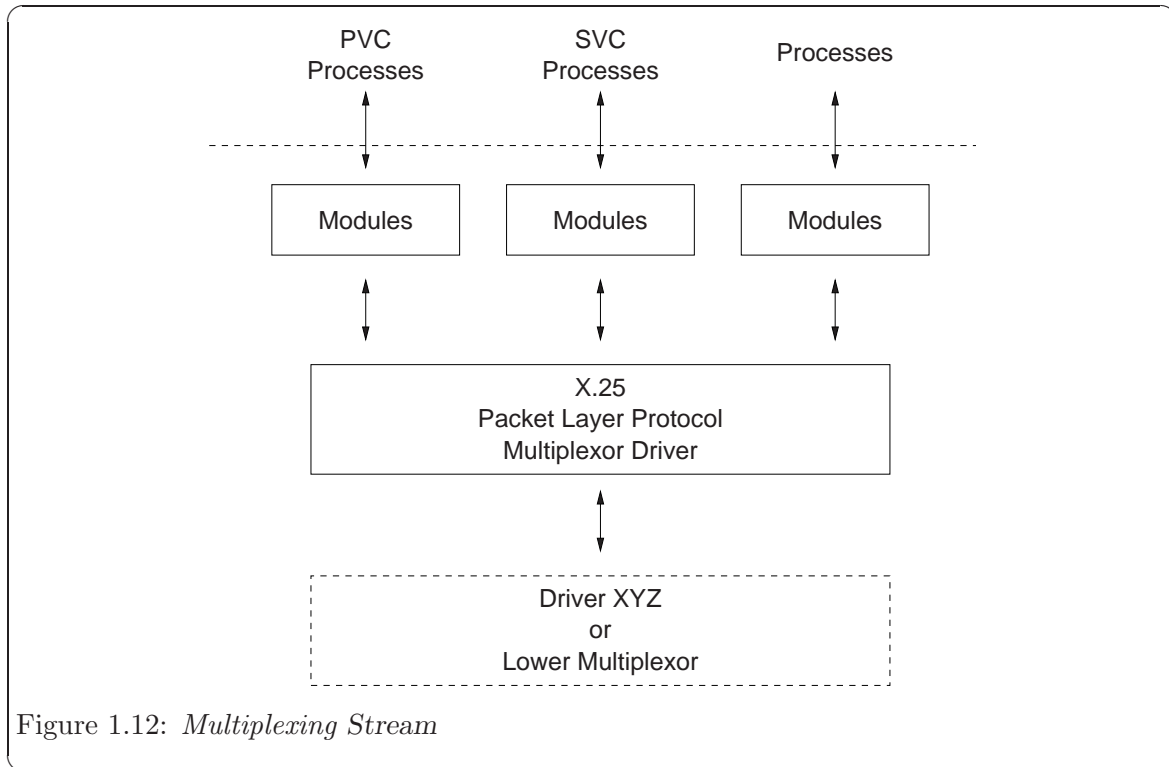


The *IP* multiplexing driver is capable of routing messages to the lower *Streams* on the basis of address and the subnet membership of each device driver. Messages received from the lower *Streams* can be discriminated and sent to the appropriate user process upper *Stream* (e.g. on the basis of, say, protocol Id). Each lower *Stream*, 'Module 1', 'Module 2', 'Driver 3', presents the same service interface to the *IP* multiplexing driver, regardless of the specific hardware or lower level communications protocol supported by the driver. For example, the lower *Streams* could all support the *Data Link Provider Interface (DLPI)*.

As depicted in [Figure 1.11](#), the *IP* multiplexing driver could have additional multiplexing drivers or modules above it. Also, 'Driver 1', 'Driver 2' or 'Driver 3' could themselves be multiplexing drivers (or replaced by multiplexing drivers). In general, multiplexing drivers are independent in the sense that it is not necessary that a given multiplexing driver be aware of other multiplexing drivers upstream of its upper *Stream*, nor downstream of its lower *Streams*.

1.5.2 Fan-In Multiplexers

[Figure 1.12](#) illustrates an example, more closely related to the *fan-in* arrangement of [Figure 1.8](#), where an *X.25 Packet Layer Protocol* multiplexing driver is used to switch messages between upper *Streams* supporting *Permanent Virtual Circuits (PVCs)* or *Switch Virtual Circuits (SVCs)* and (possibly) a single lower *Stream*.

Figure 1.12: *Multiplexing Stream*

The ability to multiplex upper *Streams* to a driver is a characteristic supported by all *STREAMS* drivers: not just *multiplexing drivers*. Each `open(2)` to a minor device node results in another upper *Stream* that can be associated with the device driver. What the *multiplexing driver* permits over the normal *STREAMS* driver is the ability to link one or more lower *Streams* (possibly containing modules and another multiplexing driver) beneath it.

1.5.3 Complex Multiplexers

When constructing multiplexers for applications, even more complicated arrangements are possible. Multiplexing over multiple *Streams* on both the upper and lower side of a *multiplexing driver* is possible. Also, a driver that provides lower multiplexing can be linked beneath a driver that provides upper multiplexing as depicted by the dashed box in [Figure 1.12](#). Each multiplexing driver can perform *upper* multiplexing, *lower* multiplexing, or both, providing a flexibility for the designer.

STREAMS provides multiplexing as a general purpose facility that is flexible in that multiplexing drivers can be stacked and linked in a wide array of complex configurations. *STREAMS* imposes few restrictions on processing within the multiplexing driver making the mechanism applicable to a many classes of applications.

Multiplexing is described in greater detail in [Chapter 10 \[Multiplexing\]](#), page 101.

1.6 Benefits of STREAMS

STREAMS provides a flexible, scalable, portable, and reusable kernel and user level facility for the development of *GNU/Linux* system communications services. *STREAMS* allows

the creation of kernel resident modules that offer standard message passing facilities and the ability for user level processes to manipulate and configure those modules into complex topologies. *STREAMS* offers a standard way for user level processes to select and interconnect *STREAMS* modules and drivers in a wide array of combinations without the need to alter *Linux* kernel code, recompile or relink the kernel.

STREAMS also assists in simplifying the user interface to device drivers and protocol stacks by providing powerful system calls for the passing of control information from user to driver. With *STREAMS* it is possible to directly implement asynchronous primitive-based service interfaces to protocol modules.

1.6.1 Standardized Service Interfaces

Many modern communications protocols define a service primitive interface between a service user and a service provider. Examples include the *ISO Open Systems Interconnect (OSI)* and protocols based on *OSI* such as *Signalling System Number 7 (SS7)*. Protocols based on *OSI* can be directly implemented using *STREAMS*.

In contrast to other approaches, such as *BSD Sockets*, *STREAMS* does not impose a structured function call interface on the interaction between a user level process or kernel resident protocol module. Instead, *STREAMS* permits the service interface between a service user and service provider (whether the service user is a user level process or kernel resident *STREAMS* module) to be defined in terms of *STREAMS* messages that represent standardized service primitives across the interface.

A service interface is defined⁷ at the boundary between neighbouring modules. The upper module at the boundary is termed the *service user* and the lower module at the boundary is termed the *service provider*. Implemented under *STREAMS*, a service interface is a specified set of messages and the rules that allow passage of these messages across the boundary. A *STREAMS* module or driver that implements a service interface will exchange messages within the defined set across the boundary and will respond to received messages in accordance with the actions defined for the specific message and the sequence of messages preceding receipt of the message (i.e., in accordance with the state of the module).

Instances of protocol stacks are formed using *STREAMS* facilities for pushing modules and linking multiplexers. For proper and consistent operation, protocol stacks are assembled so that each neighboring module, driver and multiplexer implement the same service interface. For example, a module that implements the *SS7 MTP* protocol layer, as shown in [Figure 1.13](#), presents a protocol service interface at its input and output sides. Other modules, drivers and multiplexers should only be connected at the input and output sides of the *SS7 MTP* protocol module if they provide the same interface in the symmetric role (i.e., user or provider).

It is the ability of *STREAMS* to implement service primitive interfaces between protocol modules that makes it most appropriate for implementation of protocols based on the *OSI* service primitive interface such as *X.25*, *Integrated Services Digital Network (ISDN)*, *Signalling System No. 7 (SS7)*.

⁷ See ITU-T Recommendation X.200 and ITU-T Recommendation X.210 for more information about service primitive interfaces.

1.6.2 Manipulating Modules

STREAMS provides the ability to manipulate the configuration of drivers, modules and multiplexers from user space, easing configuration of protocol stacks and profiles. Modules, drivers and multiplexers implementing common service interfaces can be substituted with ease. User level processes may access the protocol stack at various levels using the same set of standard system calls, while also permitting the service interface to the user process to match that of the topmost module.

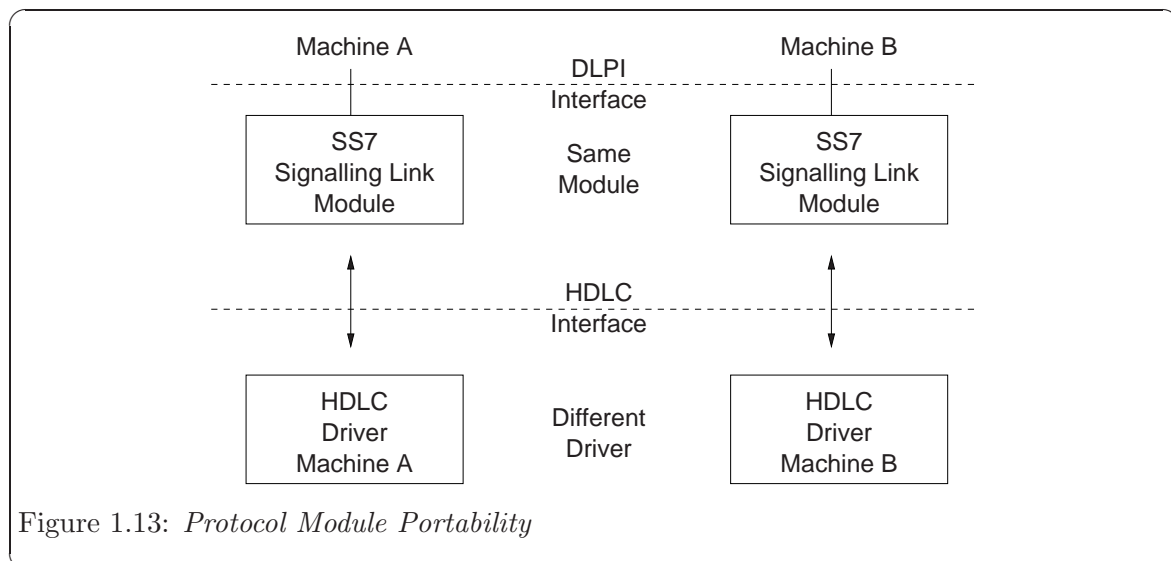
It is this flexibility that makes *STREAMS* well suited to the implementation of communications protocols based on the *OSI* service primitive interface model. Additional benefits for communications protocols include:

- User level programs use a service interface that is independent of underlying protocols, drivers, device implementation, and physical communications media.
- Communications architecture and upper layer protocols can be independent of underlying protocol, drivers, device implementation, and physical communications media.
- Communications protocol profiles can be created by selecting and connection constituent lower layer protocols and services.

The benefits of the *STREAMS* approach are protocol portability, protocol substitution, protocol migration, and module reusability. Examples provided in the sections that follow are real-world examples taken from the open source *Signalling System No. 7 (SS7)* stack implemented by the [OpenSS7 Project](#).

1.6.2.1 Protocol Portability

Figure 1.13, shows how the same *SS7 Signalling Link* protocol module can be used with different drivers on different machines by implementing compatible service interfaces. The *SS7 Signalling Link* are the *Data Link Provider Interface (DLPI)* and the *Communications Device Interface (CDI)* for *High-Level Data Link Control (HDLC)*.



By using standard *STREAMS* mechanisms for the implementation of the *SS7 Signalling Link* module, only the driver needs to be ported to port an entire protocol stack from one machine to another. The same *SS7 Signalling Link* module (and upper layer modules) can be used on both machines.

Because the *Driver* presents a standardized service interface using *STREAMS*, porting a driver from the machine architecture of ‘Machine A’ to that of ‘Machine B’ consists of changes internal to the driver and external to the *STREAMS* environment. Machine dependent issues, such as bus architectures and interrupt handling are kept independent of the primary state machine and service interface. Porting a driver from one major *UNIX* or *UNIX*-like operating system and machine architecture supporting *STREAMS* to another is a straightforward task.

With *Linux Fast-STREAMS*, *STREAMS* provides the ability to directly port a large body of existing *STREAMS* modules to the *GNU/Linux* operating system.

1.6.2.2 Protocol Substitution

STREAMS permits the easy substitution of protocol modules (or devic drivers) within a protocol stack providing a new protocol profile. When protocol modules are implemented to a compatible service interface the can be recombined and substituted, providing a flexible protocol architecture. In some circumstances, and through proper design, protocol modules can be substituted that implement the same service interface, even if they were not originally intended to be combined in such a fashion.

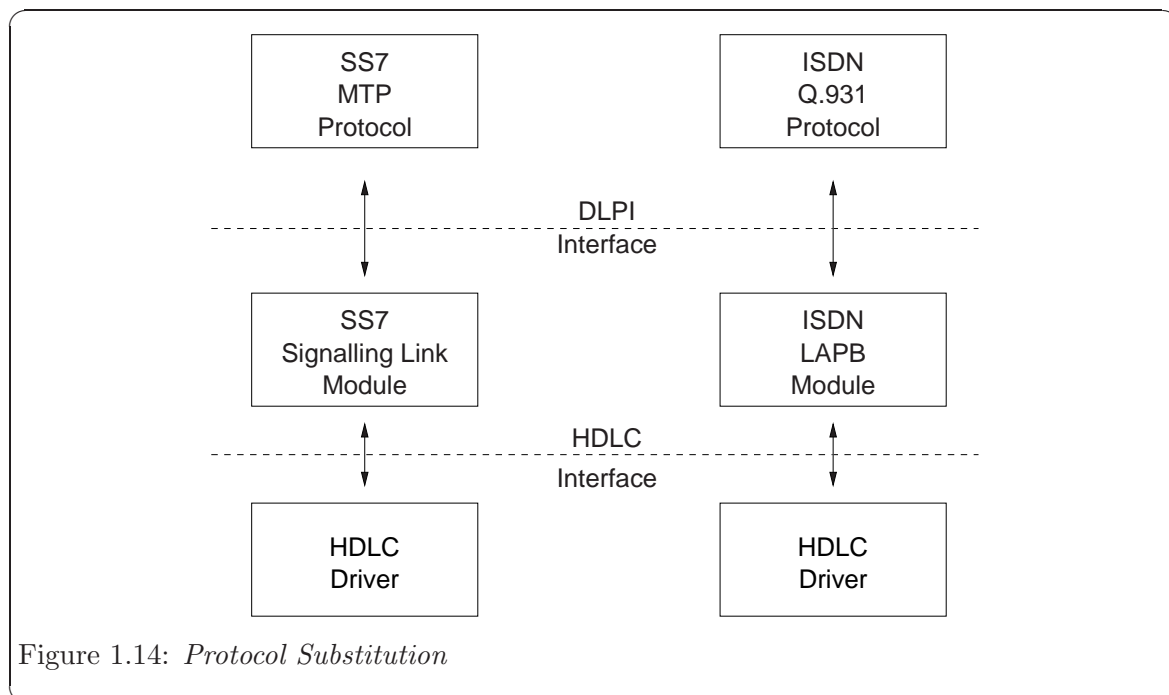


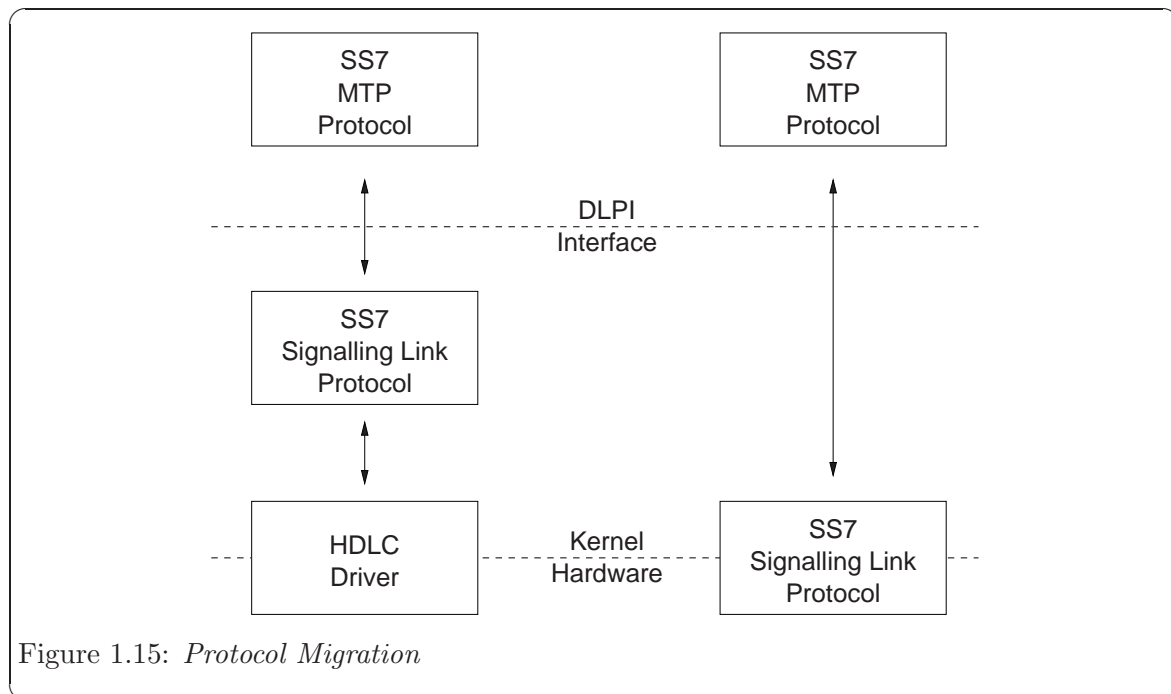
Figure 1.14 illustrates how *STREAMS* can substitute upper layer protocol modules to implement a different protocol stack over the same *HDLC* driver. As each module and driver support the same service interface at each level, it is conceivable that the resulting

modules could be recombined to support, for example, *SS7 MTP* over an *ISDN LAPB* channel.⁸

Another example would be substituting an *M2PA* signalling link module for a traditional *SS7 Signalling Link Module* to provide *SS7* over *IP*.

1.6.2.3 Protocol Migration

Figure 1.15 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 *SS7 Signalling Link* module or *SS7 Signalling Link* driver that presents the same service interface.



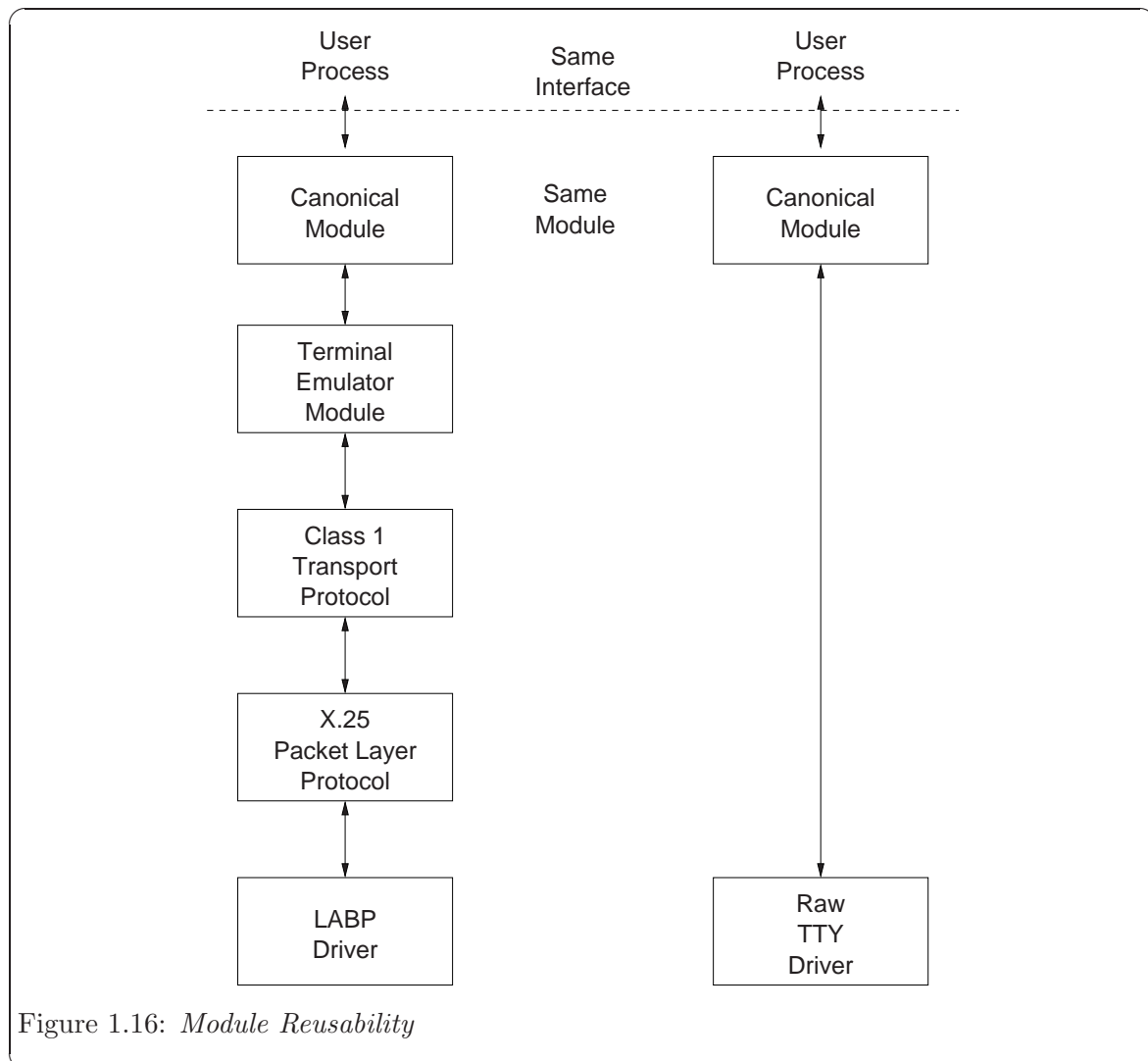
The **OpenSS7** *SS7 Stack* uses this capability also to adapt the protocol stack to front-end hardware that supports differing degrees of *SS7 Signalling Link* support in firmware. Hardware cards that support as much as a transparent bit stream can have *SS7 Signalling Data Link*, *SS7 Signalling Data Terminal* and *SS7 Signalling Link* modules pushed to provide a complete *SS7 Signalling Link* that might, on another hardware card, be mostly implemented in firmware.

By shifting functions between software and firmware, developers can produce cost effective, functionally equivalent systems over a wide range of configurations. They can rapidly incorporate technological advances. The same upper layer 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.

⁸ SS7 MTP over ISDN LAPB was originally defined under ISDN as an E-Channel.

1.6.2.4 Module Reusability

Figure 1.16 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.



2 Overview

2.1 Definitions

2.2 Concepts

2.3 Application Interface

2.4 Kernel Level Facilities

2.5 Subsystems

3 Mechanism

This chapter describes how applications programs create and interact with a *Stream* using traditional and standardized *STREAMS* system calls. General system call and *STREAMS*-specific system calls provide the interface required by user level processes when implementing user level applications programs.

3.1 Mechanism Overview

The system call interface provided by *STREAMS* is upward compatible with the traditional character device system calls.

STREAMS devices appears as character device nodes within the file system in the *GNU/Linux* system. The `open(2)` system call recognizes that a character special file is a *STREAMS* device, creates a *Stream* and associates it with a device in the same fashion as a character device.

Once open, a user process can send and receive data to and from the *STREAMS* special file using the traditional `write(2)` and `read(2)` system calls in the same manner as is performed on a traditional character device special file.

Character device input-output controls using the `ioctl(2)` system call can also be performed on a *STREAMS* special file. *STREAMS* defines a set of standard input-output control commands (see `ioctl(2p)` and `streamio(7)`) specific to *STREAMS* special files. Input-output controls that are defined for a specific device are also supported as they are for character device drivers.

With support for these general character device input and output system calls, it is possible to implement a *STREAMS* device driver in such a way that an application is unaware that it has opened and is controlling a *STREAMS* device driver: the application could treat the device in the identical manner to a character device. This makes it possible to convert an existing character device driver to *STREAMS* and make possible the portability, migration, substitution and reusability benefits of the *STREAMS* framework.

STREAMS provides *STREAMS*-specific system calls and `ioctl(2)` commands, in addition to support for the traditional character device I/O system calls and `ioctl(2)` commands.

The `poll(2)` system call¹ provides the ability for the application to poll multiple *Streams* for a wide range of events.

The `putmsg(2)` and `putpmsg(2s)` system calls provide the ability for applications programs to transfer both control and data information to the *Stream*. The `write(2)` system call only supports the transfer of data to the *Stream*, whereas, `putmsg(2)` and `putpmsg(2s)` permit the transfer of prioritized control information in addition to data.

The `getmsg(2)` and `getpmsg(2s)` system calls provide the ability for applications programs to receive both control and data information from the *Stream*. The `read(2)` system call can only support the transfer of data (and in some cases the inline control information),

¹ Although the `poll(2)` system call has been implemented in *GNU/Linux*, it was historically provided only by *STREAMS*. This is evident from the fact that `poll(2)` system can support events like `POLLRDBAND` that have no meaning outside of the *STREAMS* framework.

whereas, `getmsg(2)` and `getpmsg(2s)` permit the transfer of prioritized control information in addition to data.

Implementation of standardized service primitive interfaces is enabled through the use of the `putmsg(2)`, `putpmsg(2s)`, `getmsg(2)` and `getpmsg(2s)` system calls.

STREAMS also provides kernel level utilities and facilities for the development of kernel resident *STREAMS* modules and drivers. Within the *STREAMS* framework, the *Stream head* is responsible for conversion between *STREAMS* messages passed up and down a *Stream* and the system call interface presented to user level applications programs. The *Stream head* is common to all *STREAMS* special files and the conversion between the system call interface and message passed on the *Stream* does not have to be reimplemented by the module and device driver writer as is the case for traditional character device I/O.

3.1.1 *STREAMS* System Calls

The *STREAMS*-related system calls are:

<code>open(2)</code>	Open a <i>STREAMS</i> special file and create a new (or access an existing) <i>Stream</i> .
<code>close(2)</code>	Close a <i>STREAMS</i> special file and possibly cause the destruction of a <i>Stream</i> (i.e., on the last close of the <i>Stream</i>).
<code>read(2)</code>	Read data from an open <i>Stream</i> .
<code>write(2)</code>	Write data to an open <i>Stream</i> .
<code>ioctl(2)</code>	Control an open <i>Stream</i> .
<code>getmsg(2), getpmsg(2s)</code>	Receive a (prioritized) message at the <i>Stream head</i> .
<code>putmsg(2), putpmsg(2s)</code>	Send a (prioritized) message from the <i>Stream head</i> .
<code>poll(2)</code>	Receive notification when selected events occur on one or more <i>Streams</i> .
<code>pipe(2)</code>	Create a channel that provides a <i>STREAMS</i> -based bidirectional communication path between multiple processes.

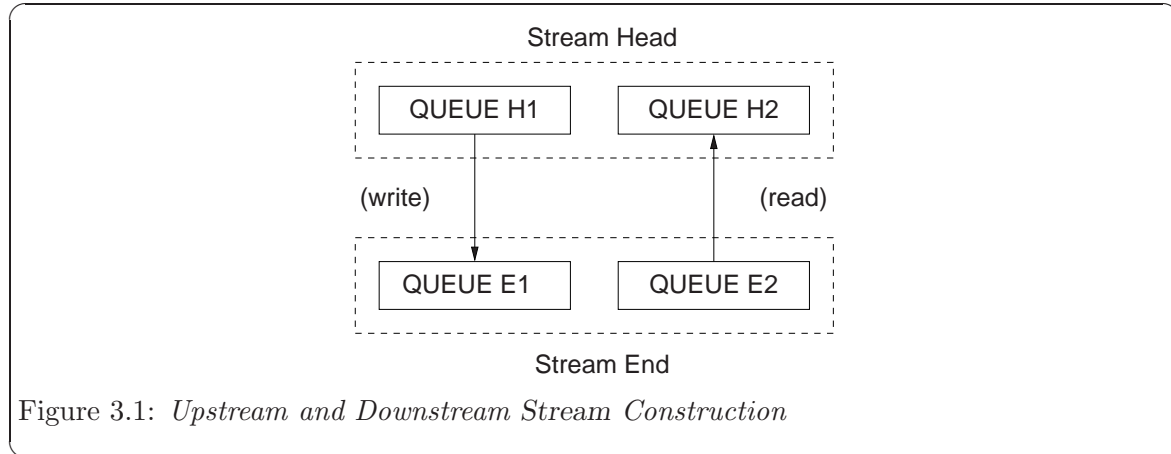
3.2 Stream Construction

STREAMS constructs a *Stream* as a double linked list of kernel data structures. Elements of the linked list are queue pairs that represent the instantiation of a *Stream head*, modules and drivers. Linear segments of link queue pairs can be connected to multiplexing drivers to form complex tree topologies. The branches of the tree are closest to the user level process and the roots of the tree are closest to the device driver.

The uppermost queue pair of a *Stream* represents the *Stream head*. The lowermost queue pair of a *Stream* represents the *Stream end* or *device driver*, *pseudo-device driver*, or another *Stream head* in the case of a *STREAMS*-based pipe.

The *Stream head* is responsible for conversion between a user level process using the system call interface and *STREAMS* messages passed up and down the *Stream*. The *Stream head* uses the same set of kernel routines available to module a driver writers to communicate with the *Stream* via the queue pair associated with the *Stream head*.

Figure 3.1 illustrates the queue pairs in the most basis of *Streams*: one consisting of a *Stream head* and a *Stream end*. Depicted are the upstream (read) and downstream (write) paths along the *Stream*. Of the uppermost queue pair illustrated, ‘H1’ is the upstream (read) half of the *Stream head* queue pair; ‘H2’, the downstream (write) half. Of the lowermost queue pair illustrated, ‘E2’ is the upstream half of the *Stream end* queue pair; ‘H1’ the downstream half.



Each queue specifies an entry point (that is, a procedure) that will be used to process messages arriving at the queue. The procedures for queues ‘H1’ and ‘H2’ process messages sent to (or that arrive at) the *Stream head*. These procedures are defined by the *STREAMS* subsystem and are responsible for the interface between *STREAMS* related system calls and the *Stream*. The procedures for queues ‘E1’ and ‘E2’ process messages at the *Stream end*. These procedures are defined by the device driver, pseudo-device driver, or *Stream head* at the *Stream end* (tail). In accordance with the procedures defined for each queue, messages are processed by the queue and typically passed from queue to queue along the linked list segment.

Figure 3.2 details the data structures involved. The data structures are the `queue(9)`, `qband(9)`, `qinit(9)`, `module_init` and `module_stat` structures.

The `queue(9)` structure is the primary data structure associated with the queue. It contains a double linked list (message queue) of messages contained on the queue. It also includes pointers to other queues used in *Stream* linkage, queue state information and flags, and pointers to the `qband(9)` and `qinit(9)` structures associated with the queue.

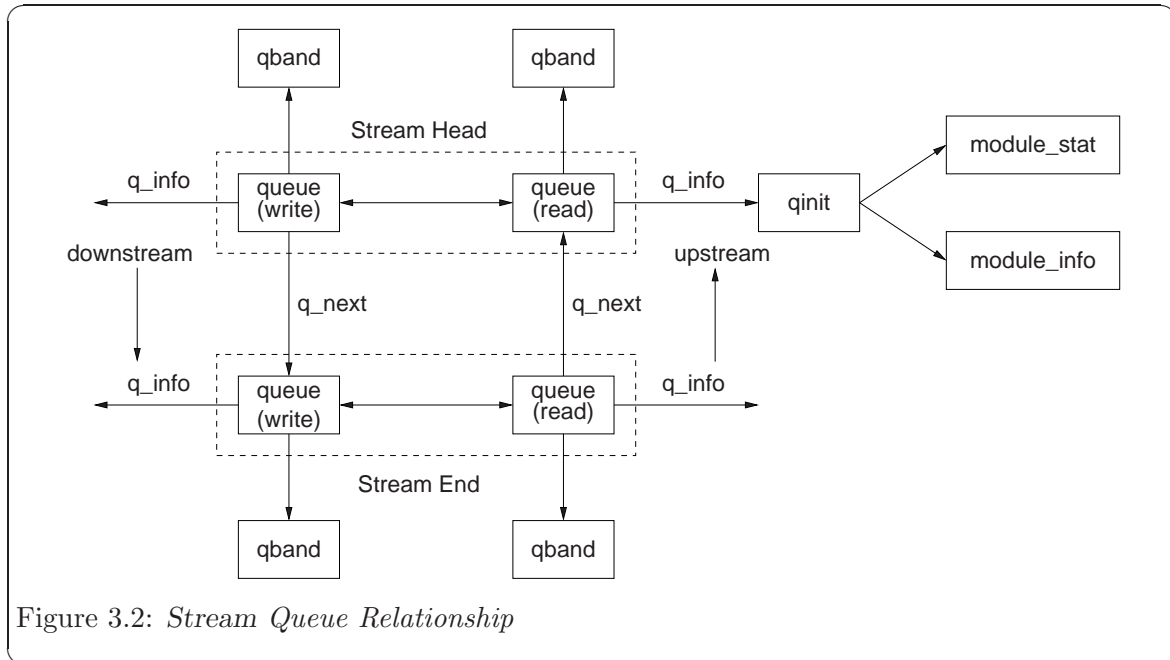
The `qband(9)` structure is used as an auxiliary structure to the `queue(9)` structure that contains state information and pointers in to the message list for each priority band within a queue (except for band ‘0’ information, which is contained in the `queue(9)` structure itself). `qband(9)` structures are linked into a list and connected to the `queue(9)` structure to which they belong.

The `qinit(9)` structure is defined by the module or driver and contains procedure pointers for the procedures associated with the queue, as well as pointers to module or driver information and initialization limits contained in the `module_info(9)` structure as well as an optional pointer to a `module_stat(9)` structure that contains collected run-time statistics

for the entire module or driver. Normally, a separate `qinit(9)` structure exists for all of the upstream and downstream instances of a queue associated with a driver or module.

The `module_info(9)` structure contains information about the module or driver, such as module identifier and module name, as well as minimum and maximum packet size and queue flow control high and low water marks. It is important to note that this structure is used only to initialize the corresponding limit values for an instance of the `queue(9)` structure. The values contained within a particular `queue(9)` structure can be changed in a running module or driver without affecting the `module_init(9)` structure. The `module_init(9)` structure is considered to be a read-only structure for the purpose of modules and drivers written for *STREAMS*.

The `module_stat(9)` structure contains runtime counts of the entry into the various procedures contained in the `qinit(9)` structure as well as a pointer to any module private statistics that need to be collected. As depicted in Figure 3.2, there is normally only one `module_stat(9)` structure per queue pair that collects statistics for the entire module or driver. *STREAMS* does not peg this counts automatically and will not manipulate this structure, even when one is attached. It is the responsibility of the module or driver writer to peg counts as required. *Linux Fast-STREAMS* does, however, provide some user level administrative tools that can be used to examine the statistics contained in this structure. The `module_stat(9)` structure is opaque to the *STREAMS* subsystem and can be read from or written to by module or driver procedures.



Note that it is possible to have a separate `qinit(9)`, `module_init(9)` and `module_stat(9)` structure for each queue in the queue pair; however, typically there are two `qinit(9)` structures and only one `module_info` and `module_stat` structure per module or driver. `qinit(9)`, `module_info` and `module_stat` structures are statically allocated by the mod-

ule or driver, and the `queue(9)` and `qband(9)` structures are dynamically allocated by *STREAMS* on demand.

All of these queue related data structures are in [Appendix A \[Data Structures\]](#), page 115 (and in the *Linux Fast-STREAMS Manual Pages*).

[Figure 3.2](#) illustrates two adjacent queue pairs with links between them in both directions on the *Stream*. When a module is opened, *STREAMS* creates a queue pair for the module and then links the the queue pair into the list. Each queue is linked to the next queue in the direction of message flow. The `q_next` member of the `queue(9)` data structure is used to perform the linkage. *STREAMS* allocates `queue(9)` structures in pairs (that is, as an array containing two `queue(9)` structures). The read-side queue of the pair is the lower ordinal and the write-side the higher. Nevertheless, *STREAMS* provides some utility functions (or macros) that assist queue procedures in locating the other queue in the pair. The *Stream head* and *Stream end* are known to procedures only a destinations toward which messages are sent.²

There are two ways for the user level process to construct a *Stream*:

1. Open a *STREAMS* device special file using the `open(2)` system call. Construction of a *Stream* with the `open(2)` system call is detailed in [Section 3.2.1 \[Opening a STREAMS Device File\]](#), page 45 and [Section 3.2.2 \[Opening a STREAMS-based FIFO\]](#), page 48 and illustrated in [Figure 3.3](#).
2. Create a *STREAMS*-based pipe using the `pipe(2)` system call. Construction of a *Stream* with the `pipe(2)` system call is detailed in [Section 3.2.3 \[Creating a STREAMS-based Pipe\]](#), page 49 and illustrated in [Figure 3.5](#).

3.2.1 Opening a STREAMS Device File

A *Stream* is constructed when a *STREAMS*-based driver file is opened using the `open(2)` system call. A *Stream* constructed in this fashion is illustrated in [Figure 3.3](#).

In the traditional *UNIX* system, a *STREAMS*-based driver file is a character device special file within the *UNIX* file system. In the *GNU/Linux* system, under *Linux Fast-STREAMS*, a *STREAMS*-based driver file is either a character device special file within a *GNU/Linux* file system, or a character device special file within the mounted *Shadow Special File System* (*specfs*). When the ‘*specfs*’ is mounted, ‘*specfs*’ device nodes can be opened directly. When the ‘*specfs*’ is not mounted, ‘*specfs*’ device nodes can only be opened indirectly via character device nodes in a *GNU/Linux* file system external to the ‘*specfs*’.

All *STREAMS* drivers (and modules) have their entry points defined by the `streamtab(9)` structure for that driver (or module). The `streamtab` structure has the following format:

² However, for the purpose of the *STREAMS* executive, most implementations cache a pointer to the *Stream head* in the `queue(9)` structure.

```

struct streamtab {
    struct qinit *st_rdinit;
    struct qinit *st_wrinit;
    struct qinit *st_muxrinit;
    struct qinit *st_muxwinit;
};

```

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. For a multiplexing driver, the `st_muxrinit` and `st_muxwinit` point to the `qinit` structures for the lower side of the multiplexing driver. For a regular non-multiplexing driver these members are NULL.

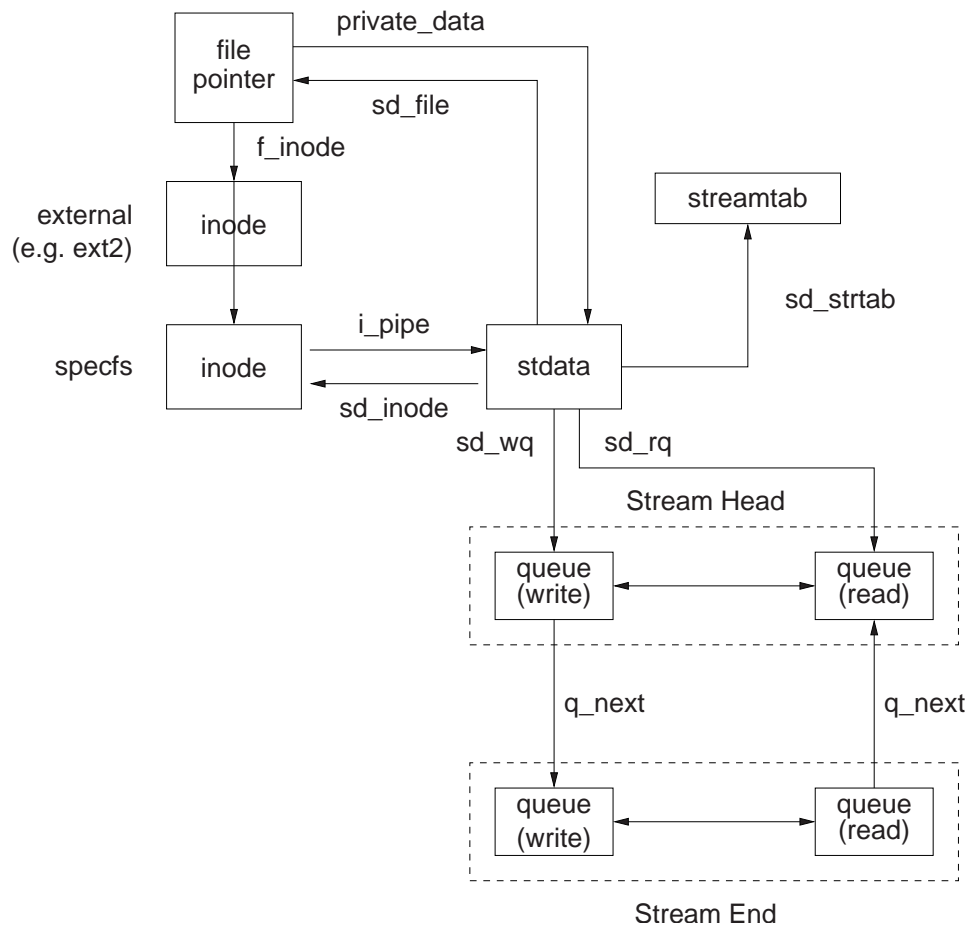


Figure 3.3: *Opened STREAMS-based Driver*

3.2.1.1 First Open of a Stream

When a *STREAMS*-based file is opened, a new *Stream* is created if one does not already exist for the file, or if the `D_CLONE` flag is set for the file indicating that a new *Stream* is to be created on each open of the file. First, a file descriptor is allocated in the process'

file descriptor table, a file pointer is allocated to represent the opened file. The file pointer is initialized to point to the `inode` associated with the character special file in the external file system (see `f_inode` in Figure 3.3). This `inode` is of type character special (`S_IFCHR`). The Linux kernel recognizes the `inode` as a character special file and invokes the character device open routine in *Linux Fast-STREAMS*. This `inode` is equivalent to the `vnode` used by *UNIX System V Release 4.2*.

Linux Fast-STREAMS uses the major and minor device numbers associated with the character special file to locate an `inode` within the *Shadow Special File System (specfs)* that is also provided by *Linux Fast-STREAMS*, and the `f_inode` pointer of the file pointer is adjusted to point directly to this ‘specfs’ `inode`. This ‘specfs’ `inode` is equivalent to the common `snode` used by *UNIX System V Release 4.2*.

Next, a *Stream header* is created from a `stdata(9)` data structure and a *Stream head* is created from a pair of `queue` structures. The content of the `stdata` data structure is initialized with predetermined *STREAMS* values applicable to all character special *Streams*. The content of the `queue` data structures in the *Stream head* are initialized with values from the `streamtab` structure statically defined for *Stream heads* in the same manner as any *STREAMS* module or driver.

The `inode` within the ‘specfs’ contains *STREAMS* file system dependent information. This `inode` corresponds to the common `snode` of *UNIX System V Release 4.2*. The `sd_inode` field of the `stdata` structure is initialized to point to this `inode`. The `i_pipe` field of the `inode` data structure is initialized to point to the *Stream header* (`stdata` structure), thus there is a forward and backward pointer between the *Stream header* and the `inode`.

The `private_data` member of the `file` pointer is initialized to point to the *Stream header* and the `sd_file` member of the `stdata` structure is initialized to point to the `file` pointer.

After the *Stream header* and *Stream head* queue pair is allocated and initialized, a `queue` structure pair is allocated and initialized for the driver. Each `queue` in the queue pair has its `q_init` pointer initialized to the corresponding `qinit` structure defined in the driver’s `streamtab`. Limit values in each `queue` in the pair are initialized the queue’s `module_init` structure, now accessible via the `q_init` pointer in the `queue` structure and the `qi_mininfo` pointer in the `qinit` structure.

The `q_next` pointers in each `queue` structure 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` pointers at the ends of the *Stream* are set to `NULL`. Finally, the driver open procedure (accessible via the `qi_qopen` member of the `qinit` structure for the read-side queue) is called.

3.2.1.2 Subsequent Open of a Stream

When the *Stream* has already been created by a call to `open(2)` and has not yet been destroyed, that is, on a subsequent open of the *Stream*, and the *STREAMS* driver is not marked for clone open with the `D_CLONE` flag in the `cdevsw(9)` structure, the only actions performed are to call the driver’s `open` procedure and the `open` procedures of all pushable modules present on the already existing *Stream*.

3.2.2 Opening a STREAMS-based FIFO

A *STREAMS*-based FIFO *Stream* is also constructed with a call to `open(2)`. A *Stream* constructed in this fashion is illustrated in Figure 3.4.

A *STREAMS*-based FIFO appears as a FIFO special file within a *GNU/Linux* file system, as a character special file within a *GNU/Linux* file system, or as a FIFO special file within the *Shadow Special File System (specfs)*.³

Figure 3.4 illustrates an *STREAMS*-based FIFO that has been opened and a *Stream* created.

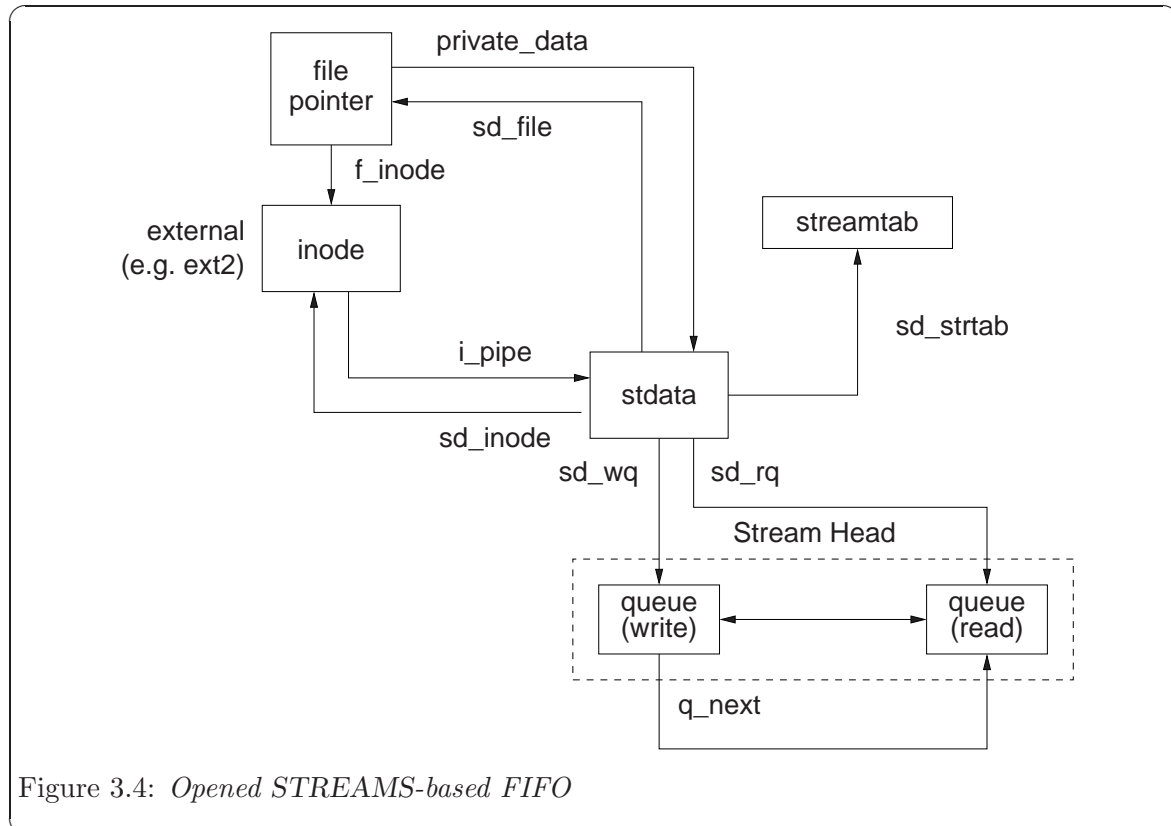


Figure 3.4: *Opened STREAMS-based FIFO*

The sequence of events that cause the creation of a *Stream* when a *STREAMS*-based FIFO is opened using the `open(2)` system call are the same as that for regular *STREAMS* device special files with the following differences:

1. When the *Stream header* (`stdata` structure) is created, it is attached to the external *GNU/Linux* file system `inode` instead of an `inode` within the *Shadow Special File System (specfs)*. This is also true of the `file pointer`: that is, the `file pointer` refers

³ This is different than the situation in the *UNIX System V Release 4.2* system and other *UNIX* variants in the following respects: In *SVR 4.2* all FIFOs are *STREAMS*-based. In other *UNIX* implementations FIFOs are either *SVR 3.2*-style or, in some systems, optionally *STREAMS*-based. In *SVR 4.2* FIFOs are FIFO special files. In other *UNIX* implementations (and in *LiS*), FIFOs are character special files. Under *GNU/Linux*, system FIFOs are by default *SVR 3.2*-style FIFOs. To achieve the greatest possible degree of compatibility, *Linux Fast-STREAMS* provides the option of making all *GNU/Linux* system FIFOs *STREAMS*-based, and also provides a character special file implementation of *STREAMS*-based FIFOs.

to the external file system `inode` instead of a ‘`specfs`’ `inode`. The result is illustrated in Figure 3.4.

2. The *Stream header* (`stdata` structure) is initialized with limits and values appropriate for a *STREAMS*-based FIFO rather than a regular *STREAMS* driver. This is because the behaviour of a *STREAMS*-based FIFO *Stream head* must be somewhat different from a regular *STREAMS* driver to be compliant with *POSIX*.⁴
3. No driver queue pair is created or attached to the *Stream*. The *Stream head* write-side queue `q_next` pointer is set to the read-side queue as illustrated in Figure 3.4.

Aside from these differences, opening a *STREAMS*-based FIFO is structurally equivalent to opening a regular *STREAMS* driver. The similarity makes it possible to also implement *STREAMS*-based FIFOs as character special files.

3.2.3 Creating a STREAMS-based Pipe

A *Stream* is also constructed when a *STREAMS*-based pipe is created using the `pipe(2)` system call.⁵ A *Stream* constructed in this fashion is illustrated in Figure 3.5.

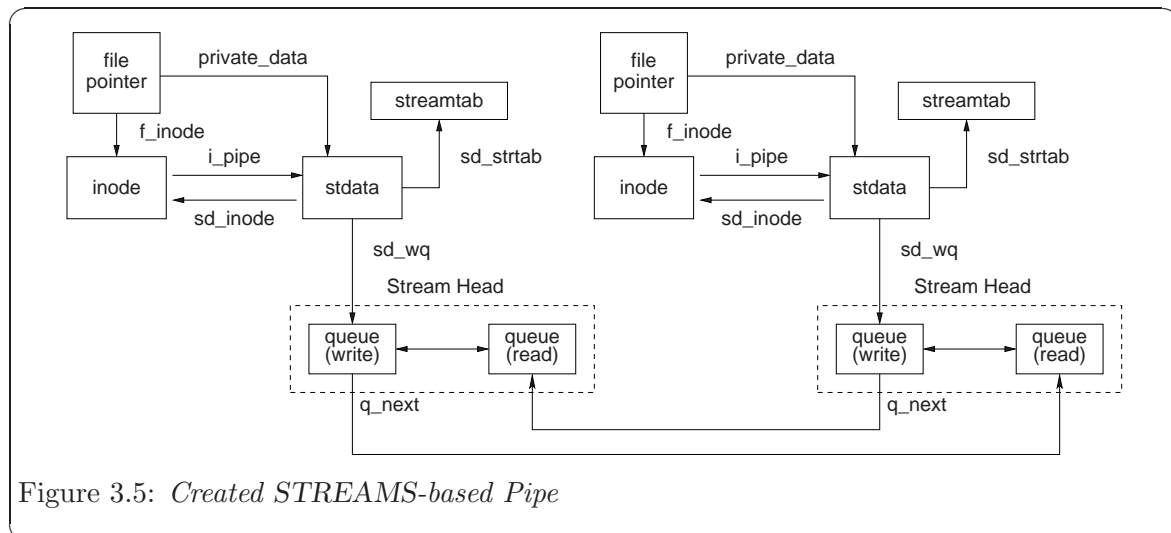


Figure 3.5: Created *STREAMS*-based Pipe

Pipes have no `inode` in an external *GNU/Linux* file system that can be opened with the `open(2)` system call and, therefore, they must be created with a call to `pipe(2)`.⁶ When the `pipe(2)` system call is executed, two *Streams* are created. The construction of each *Stream* is similar to that when a *STREAMS* driver is opened with the following differences:

⁴ For example, a FIFO opened read-only will block waiting for another process to open the FIFO for writing.

⁵ Note that, by default, *GNU/Linux* system pipes obtained with the `pipe(2)` system call are SVR 3.2-style unidirectional pipes. *Linux Fast-STREAMS* provides a `pipe(2)` library function in the ‘`libstreams`’ library that can be used to override the normal `pipe(2)` system call for some applications programs. Also, *Linux Fast-STREAMS* provides the option of overriding all system pipes returned by the `pipe(2)` system call to be bidirectional *STREAMS*-based pipes.

⁶ Some *UNIX* implementations, notably *UnixWare*, provide the ability to open two character special files and associate them together into a *STREAMS*-based pipe (see `sfx(4)`). In that case, opening each end of a *STREAMS*-based pipe is no different than opening a regular *STREAMS* driver.

- Instead of creating one process file table entry and one `file` pointer, as was the case for regular *STREAMS* drivers, `pipe(2)` creates two file table entries (file descriptors) and two `file` pointers, as shown in Figure 3.5.
- Because a character special device is not being opened, there is no `inode` in an external file system, so *STREAMS* allocated two `inodes` from the ‘`specfs`’.⁷ Each `inode` has a file type of `S_IFIFO`. The `file` pointer and `stdata` structure for each *Stream head* is attached to one of these `inodes`.
- When the *Stream head* associated with each file descriptor is initialized, the `stdata` structure is initialized with values appropriate for a *STREAMS*-based pipe instead of a regular *Stream*.⁸
- Instead of creating a driver queue pair for the *Stream*, the `q_next` pointer for the write-side queue of each *Stream head* is initialized to point to the read-side queue of the other *Stream head*. This is illustrated in Figure 3.5.

3.2.4 Adding and Removing Modules

When a *Stream* has been constructed, modules can be inserted into the *Stream* between the *Stream head* and the *Stream end* (or between the *Stream head* and the midpoint of a *STREAMS*-based pipe or FIFO.) Addition (or pushing) of modules is accomplished by inserting the module into the *Stream* immediately below the *Stream head*. Removal (or popping) of modules is accomplished by deleting the module immediately below the *Stream head* from the *Stream*.

When a module is pushed onto a *Stream*, the module’s `open` procedure is called for the newly inserted queue pair. When a module is popped from the *Stream*, the module’s `close` procedure is called prior to deleting the queue pair from the *Stream*.

Modules are pushed onto an open *Stream* by issuing the `I_PUSH(7) ioctl(2)` command on the file descriptor associated with the open *Stream*. Modules are popped from a *Stream* with the `I_POP(7) ioctl(2)` command on the file descriptor associated with the open *Stream*.

`I_PUSH` and `I_POP` allow a user level process to dynamically reconfigure the ordering and type of modules on a *Stream* to meet any requirement.

3.2.4.1 Pushing Modules

When the *Stream head* receives an `I_PUSH ioctl` command, *STREAMS* locates the module’s `streamtab` entry and creates a new queue pair to represent the instance of the module. Each queue in the pair is initialized in a similar fashion as for drivers: the `q_init` pointers are initialized to point to the `qinit` structures of the module’s `streamtab`, and the limit values are initialized to the values found in the corresponding `module_init` structures.

Next, *STREAMS* positions the module’s queue pair in the *Stream* immediately beneath the *Stream head* and above the driver and all existing modules on the *Stream*. Then the

⁷ Some UNIX implementations, and UNIX System V Release 4, provide a separate file system, the ‘`pipefs`’, upon which `vnodes` are created. In a similar fashion, GNU/Linux SVR 3.2-style system pipes also allocates `inodes` from a ‘`pipefs`’ file system.

⁸ Examples of differences include that pipes issue `SIGPIPE` when the *Stream* encounters an error, that is, the `SNDPIPE` write option is enabled, and pipe cannot send zero-length data by default, that is, the `SNDZERO` write option is disabled. Both of these are the reverse for a regular *Stream*.

module's `open` procedure is called for the queue pair. (The `open` procedure is located in the `qi_qopen` member of the `qinit` structure associated with the read-side queue.)

Each push of a module onto a *Stream* results in the insertion of a new queue pair representing a new instance of the module. If a module is (successfully) pushed twice on the same *Stream*, two queue pairs and two instances of the module will exist on the *Stream*.

To assist in identifying misbehaving applications programs that might push the same set of modules in an indefinite loop, swallowing an excessive amount of system resources, *STREAMS* imposes a limit on the number of modules that can be pushed on a given *Stream* to a practical number. The number is limited by the *NSTRPUSH* kernel parameter (see [Appendix E \[Configuration\], page 123](#)) which is set to either '16' or '64' on most systems.

Once an instance of a module is pushed on a *Stream*, its `open` procedure will be called each time that the *Stream* is reopened.

3.2.4.2 Popping Modules

When the *Stream head* receives a `I_POP ioctl` command, *STREAMS* locates the module directly beneath the *Stream head* and calls its `close` procedure. (The `close` procedure is located by the `qi_qclose` member in the `qinit` structure associated with the module instance's read-side queue.) Once the `close` procedure returns, *STREAMS* deletes the queue pair from the *Stream* and deallocates the queue pair.

3.2.5 Closing the Stream

Relinquishing the last reference to a *Stream* dismantles the *Stream* and deallocates its components. Normally, the last direct or indirect call to `close(2)` for a *Stream* results in the *Stream* being dismantled in this fashion.⁹ Calls to `close(2)` before the last close of a *Stream* will not result in the dismantling of the *Stream* and no module or driver `close` procedure will be called on closes prior to the last close of a *Stream*.

Dismantling a *Stream* consists of the following sequence of actions:

1. If the *Stream* is a *STREAMS*-based pipe and the other end of the pipe is not open by any process, but is named (i.e., mounted by `fattach(3)`), then the named end of the pipe is detached as with `fdetach(3)` and then the *Stream* is dismantled.
2. If the *Stream* is a multiplexing driver, dismantling a *Stream* first consists of unlinking any *Streams* that remain temporarily linked (by a previous `I_LINK` command) under the multiplexing driver using the *control stream* being closed. Unlinking of temporary links consists of issuing an `M_IOCTL` message to the driver indicating the `I_UNLINK` operation and entering an uninterrupted wait for an acknowledgement. Waiting for acknowledgement to the `M_IOCTL` command can cause the close to be delayed. If unlinking any temporary links results in the last reference being released to the now unlinked *Stream*, that *Stream* will be dismantled before proceeding.

⁹ Exceptions are when the *Stream* has been named with `fattach(8)`, that is, it is still *mounted*, or when the *Stream* is still linked under a multiplexing driver.

3. Each module that is present on the *Stream* being dismantled will be popped from the *Stream* by calling the module's `close` procedure and then deleting the module instance queue pair from the *Stream*.
4. If a driver exists on the *Stream* being dismantled, the driver's `close` procedure is called and then the *Stream end* queue pairs are deallocated.

If the *Stream* invoking the chain of events that resulted in the dismantling of a *Stream* is open for blocking operation (neither `O_NDELAY` nor `O_NONBLOCK` were set), no signal is pending for the process causing dismantling of the *Stream*, and there are messages on the module or driver's write-side queue, *STREAMS* may wait for an interval for the messages to drain before calling the module or driver's `close` procedure. The maximum interval to wait is traditionally '15' seconds. If any of these conditions are not met, the module or driver is closed immediately.

When each module or driver queue pair is deallocated, any messages that remain on the queue are flushed prior to deallocation. Note that *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 before it returns from its `close` procedure.

5. The queue pair associated with the *Stream head* is closed¹⁰ and the queue pair and *Stream header* (`stdata` structure) are deallocated and the associated `inode`, `file` pointer, and file descriptors are released.

3.2.6 Stream Construction Example

This *Streams* construction example builds on the previous example (see [Listing 1.1 in Section 1.3 \[Basic Streams Operations\]](#), page 17), by adding the pushing of a module onto the open *Stream*.

3.2.6.1 Inserting Modules

This example demonstrates the ability of *STREAMS* to push modules, not available with traditional character devices. The ability to push modules onto a *Stream* allows the independent processing and manipulation of data passing between the driver and user level process. This example is of a character conversion module is given a command and a string of characters by the user. Once this command is received, the character conversion module examines all character passing through it for an occurrence of the characters in the command string. When an instance of the string is discovered in the data path, the requested command action is performed on matching characters.

The declarations for the user program are shown in [Listing 3.1](#).

¹⁰ Note that the messages are not queued on the *Stream head* write-side queue and so no delay in closing the *Stream head* queue pair is considered.


```

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/uio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stropts.h>

#define    BUFLLEN    1024

/*
 * These defines 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 striocctl striocctl;

```

Listing 3.1: *Inserting Modules Example*

As in the previous example of [Listing 1.1](#), first a *Stream* is opened using the `open(2)` system call. In this example, the *STREAMS* device driver is `‘/dev/streams/comm/01’`.

```

if ((fd = open("/dev/streams/comm/01", O_RDWR)) < 0) {
    perror("open failed");
    exit(1);
}

```

Listing 3.2: *Inserting Modules Example (cont'd)*

Next, the character conversion module (named `‘chconv’`) is pushed onto the open *Stream* using the `I_PUSH(7) ioctl(2)` command.

```

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

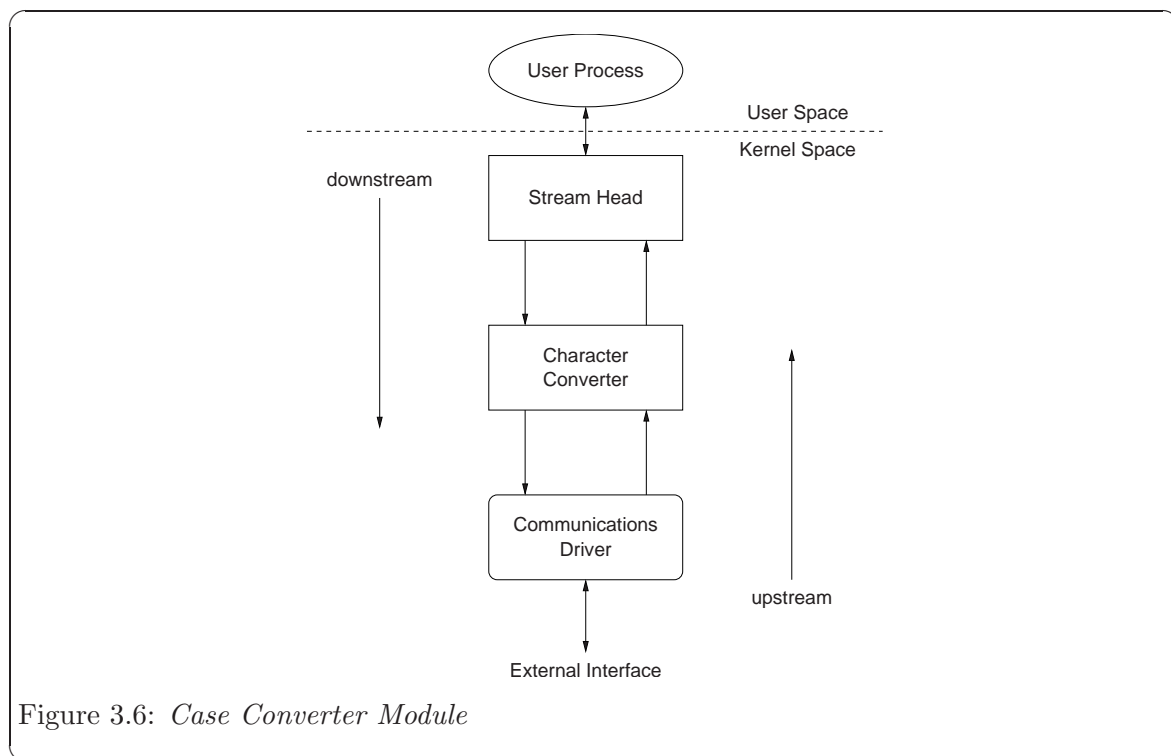
```

Listing 3.3: *Inserting Modules Example (cont'd)*

The difference in creating an instance of a *STREAMS* driver and module are illustrated in [Listing 3.2](#) and [Listing 3.3](#). An instance of a driver is created with the `open(2)` system call, and each driver requires at least one device node in a file system for access. Naming of device nodes follow device naming conventions. On the other hand, an instance of a module is created with the `I_PUSH(7) ioctl(2)` command. No file system device node is required. Naming of modules is separate from any file system considerations, and are chosen by the

module writer. The only restrictions on a module name is that it be less than `FMNAMESZ` in length, and that it be unique.

When successful, the `I_PUSH(7) ioctl(2)` call directs *STREAMS* to locate and insert the *STREAMS* module named ‘`chconv`’ onto the *Stream*. If the ‘`chconv`’ module has not been loaded into the *Linux* kernel, *Linux Fast-STREAMS* will attempt to demand load the kernel module named ‘`streams-chconv`’. Once the ‘`chconv`’ *STREAMS* module is loaded in the kernel, *STREAMS* will create a queue pair for the instance of the module, insert it into the *Stream* beneath the *Stream head*, and call the module’s `open` procedure. If the module’s `open` procedure returns an error (typically only `[ENXIO]`), that error will be returned to the `ioctl(2)` call. If the module’s `open` procedure is successful, it (and the `ioctl(2)` call), return ‘0’. The resulting *Stream* configuration is illustrated in [Figure 3.6](#).



Modules are always pushed and popped from the position immediately beneath the *Stream head* in the manner of a push-down stack. This results in a *Last-In-First-Out (LIFO)* order of modules being pushed and popped. For example, if another module were to be pushed on the *Stream* illustrated in [Figure 3.6](#), it would be placed between the *Stream head* and the *Character Converter* module.

3.2.6.2 Module and Driver Control

The next steps in this example are to pass control information to the module to tell it what command to execute on which string of characters. A sequence that achieves this is shown in [Listing 3.4](#). The sequence makes use of the `I_STR(7) ioctl(2)` command for *STREAMS* special files.

```

/* 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);
}

```

Listing 3.4: *Module and Driver Control Example*

There exist two methods for controlling modules and drivers using the `ioctl(2)` system call:

Transparent

In a transparent `ioctl(2)` call, the *cmd* argument to the call is the command issued to the module or device, and the *arg* argument is specific to the command and defined by the receiver of the command. This is the traditional method of controlling character devices and can also be supported by a *STREAMS* module and driver.

I_STR

In an `I_STR` `ioctl(2)` call, the *cmd* argument to the call is `I_STR` and the *arg* argument of the call is a pointer to a `striocctl` structure (defined in ‘`sys/stropts.h`’) describing the particulars of the call. This method is specific to *STREAMS* special files.

It is this later method that illustrated in [Listing 3.4](#).

The `striocctl` structure, defined in ‘`sys/stropts.h`’, has the following format:

```

struct striocctl {
    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 */
};

```

<i>ic_cmd</i>	identifies the command intended for a module or driver,
<i>ic_timeout</i>	specifies the number of seconds an <code>I_STR</code> request should wait for an acknowledgement 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 [Listing 3.4](#), two commands are issued to the character conversion module, **XCASE** and **DELETE**.¹¹

To issue the example **XCASE** command, *ic_cmd* is set to the command, **XCASE**, and *ic_dp* and *ic_len* are set to the the string ‘AEIOU’. Upon receiving this command, the example module will convert uppercase vowels to lowercase in the data subsequently passing through the module. *ic_timeout* is set to zero to indicated that the default timeout (‘15’ seconds) should be used if no response is received.

To issue the example **DELETE** command, *ic_cmd* is set to the command, **DELETE**, and *ic_dp* and *ic_len* are set to the the string ‘xX’. Upon receiving this command, the example module will delete all occurrences of the characters ‘X’ and ‘x’ from data subsequently passing through the module. *ic_timeout* is set to zero to indicated that the default timeout (‘15’ seconds) should be used if no response is received.

Once issued, the *Stream head* takes an **I_STR ioctl(2)** command and packages its contents into a *STREAMS* message consisting of an **M_IOCTL** block and a **M_DATA** block and passes it downstream to be considered by modules and drivers on the *Stream*. The *ic_cmd* and *ic_len* values are stored in the **M_IOCTL** block and the data described by *ic_dp* and *ic_len* are copied into the **M_DATA** block. Each module, and ultimately the driver, examines the *ioc_cmd* filed in the **M_IOCTL** message to see if the command is known to it. If the command is unknown to a module, it is passed downstream for consideration by other modules on the *Stream* or for consideration by the driver. If the command is unknown to a driver, it is negatively acknowledged and a error is returned from the **ioctl(2)** call.

The user level process calling **ioctl(2)** with the **I_STR(7)** command will block awaiting an acknowledgement. The calling process will block up to *ic_timeout* seconds waiting for a response. If *ic_timeout* is ‘0’, it indicates that the default timeout value (typically ‘15’ seconds) should be used. If *ic_timeout* is ‘-1’, it indicates that an infinite timeout should be used. If the timeout occurs, the **ioctl(2)** command will fail with error **[ETIME]**. Only one process (thread) can be executing an **I_STR(7) ioctl(2)** call on a given *Stream* at time. If an **I_STR** is being executed when another process (or thread) issues an **I_STR** of its own, the process (or thread) will block until the previous **I_STR** operation completes. However, the process (or thread) will not block indefinitely if *ic_timeout* is set to a finite timeout value.

When successful, the **I_STR** command returns the value defined by the command operation itself, and also returns any information to be returned in the area pointed to by *ic_dp* on the call. The *ic_len* member is ignored for the purposes of returning data, and it is the caller’s responsibility to ensure that the buffer pointed to by *ic_dp* is large enough to hold the returned data.

3.2.6.3 Stream Dismantling with Modules

As shown in [Listing 3.5](#), the remainder of this example follows the example in [Listing 1.1](#) in [Section 1.3 \[Basic Streams Operations\]](#), page 17: data is read from the *Stream* and then echoed back to the *Stream*.

¹¹ These commands are fictitious.

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

Listing 3.5: *Module and Driver Control Example (cont'd)*

The `exit(2)` system call in Listing 3.5 will result in the dismantling of the *Stream* as it is closed. However, in this example, when the *Stream* is closed with the ‘`chconv`’ module still present on the *Stream*, the module is automatically popped as the *Stream* is dismantled.

Alternatively, it is possible to explicitly pop the module from the *Stream* using the `I_POP(7)` `ioctl(2)` command. The `I_POP` command removes the module that exists immediately below the *Stream head*. It is not necessary to specify the module to be popped by name: whatever module exists just beneath the *Stream head* will be popped.

3.2.6.4 Stream Construction Example Summary

This example provided illustration of the ability of *STREAMS* to modify the behaviour of a driver without the need to modify driver code. A *STREAMS* module was pushed that provided the extended behaviour independent of the underlying driver. The `I_PUSH` and `I_POP` commands used to push and pop *STREAMS* modules were also illustrated by the example.

Many other `streamio(7)` `ioctl` commands are available to the applications programmer to manipulate and interrogate configuration and other characteristics of a *Stream*. See `streamio(7)` for details.

4 Processing

Each module or driver queue pair has associated with it **open** **close** and optionally **admin** procedures. These procedures are specified by the *qi_qopen*, *qi_qclose* and *qi_qadmin* function pointers in the **qinit**(9) structure associated with the read-side **queue**(9) of the queue pair. The **open** and **close** procedures was the focus of previous chapters.

Each **queue**(9) in a module or driver queue pair has associated with it a **put** and optional **service** procedure. These procedures are specified by the *qi_putp* and *qi_srvp* function pointers in the **qinit**(9) structure associated with each **queue**(9) in the queue pair. The **put** and **service** procedures are responsible for the processing of messages the implementation of flow control, and are the focus of this chapter.

4.1 Procedures

The **put** and **service** procedures associated with a given **queue**(9) in a module queue pair are responsible for the processing of messages entering and leaving the queue. Processing within these procedures is performed according to the message type of the message being processed. Messages can be modified, queued, passed in either direction on a *Stream*, freed, copied, duplicated, or otherwise manipulated. In processing for typical filter module, a resulting message is normally passed along the *Stream* in the same direction it was travelling when it was received.

A queue must always have a **put** procedure. The **put** procedure will be invoked when messages are passed to the queue from an upstream or downstream module. A **put** procedure will either process the message immediately, or place the message on its queue awaiting later processing by the module or driver's **service** procedure.

Optionally, a queue can also have an associated **service** procedure. The **service** procedure is responsible for processing the backlog of any queued messages from the message queue.

With both a **put** and **service** procedure it is possible to tune performance of a module or driver by performing actions required immediately from the **put** procedure while performing actions that can be deferred from the **service** procedure. The **service** procedure provides for the implementation of flow control and can also be used to promote bulk processing of messages.

The **put** and particularly the **service** procedures are not directly associated with any user level process. They are kernel level coroutines that normally run under the context of the *STREAMS Scheduler* kernel thread.¹

4.1.1 Put Procedure

The **put** procedure is invoked whenever a message is passed to a queue. A message can be passed to a queue using the **put**(9), **putnext**(9), **putctl**(9), **putctl1**(9), **putctl2**(9), **putnextctl**(9), **putnextctl1**(9), **putnextctl2**(9), **qreply**(9) *STREAMS* utilities. The

¹ Under some restricted circumstances, a module or driver **put** procedure is run under a user context when invoked from a *Stream head*, or under an interrupt service routine or software interrupt when invoked from a *Stream end (driver)*.

Stream head, modules and drivers use these utilities to deliver messages to a queue.² Invoking the **put** procedure of a queue with one of these utilities is the only accepted way of passing a message to a queue.³

A queue's **put** procedure is specified by the *qi_putp* member of the **qinit(9)** structure associated with the **queue(9)**. This is illustrated in Figure 4.1. In general, the read- and write-side queues of a module or driver have different **qinit(9)** structures associated with them as there are differences in upstream and downstream message processing; however, it is possible for read- and write-side queues to share the same **qinit(9)** structure.

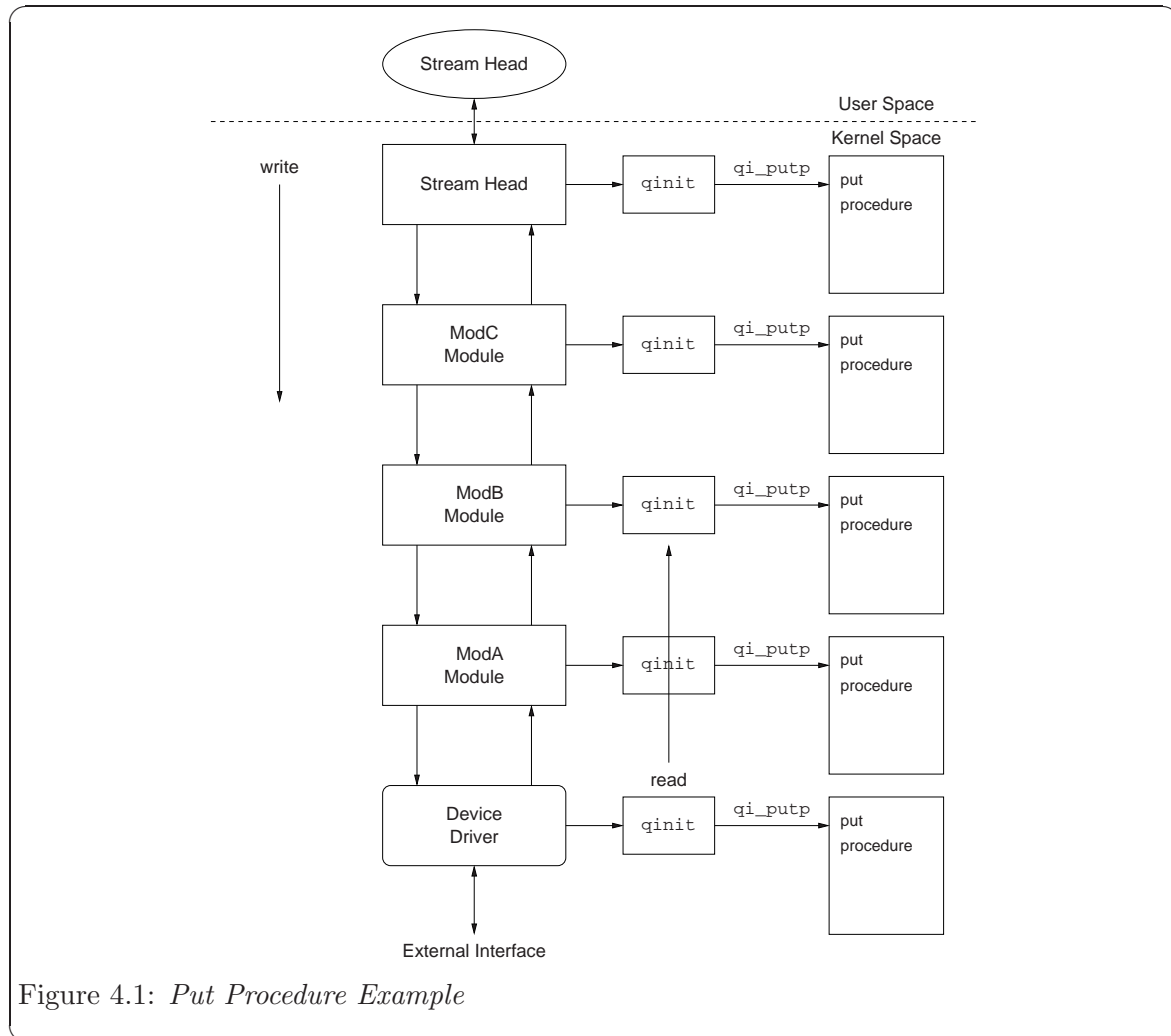


Figure 4.1: *Put Procedure Example*

The **put** procedure processes a message immediately or places it onto the message queue for later processing (generally by the **service** procedure). Because the **put** procedure is

² The *qi_putp* procedure should not be called directly.

³ In special circumstances, such as in a *Stream end* or driver, it is possible to use *putq(9)* to place a message on a queue to be later retrieved by the driver's **service** procedure; however, this practice is the same as setting the driver's *qi_putp* pointer to *putq(9)*.

invoked before any queueing takes place, it provides a processing point at which the module or driver can take actions on time critical messages. **put** procedures are executed a higher priority than **service** procedures. **put** procedures in the upstream direction may even be executed with interrupts disabled.

As illustrated in Figure 4.1, when a queue's **put** procedure is invoked by an adjacent queue's **put** procedure (e.g. using `putnext(9)`), the `qi_putp` member of the queue's associated `qinit(9)` structure is invoked by *STREAMS* as subroutine call.

When a number of modules are present in a *Stream*, as illustrated in Figure 4.1, each successive direct invocation of a **put** procedure is nested inside the others. For example, if the **put** procedure on the read-side of the driver is invoked by calling `put(9)` from the driver's interrupt service routine, and then each successive **put** procedure calls `putnext(9)`, by the time that the message reaches the *Stream head*, the driver, 'ModA', 'ModB', 'ModC', and the *Stream head* **put** procedures will be nested within another.

The advantage of this approach is that **put** processing is invoked sequentially and immediately. A disadvantage of this approach is that, if there are additional stack frames nested in each **put** procedure, the interrupt service routine stack limits can be exceeded, causing a kernel crash. This is also the case for normal (non-ISR) operation and the kernel stack limits might be exceeded if excessive nesting of **put** procedures occurs.⁴

The driver and module writers need to be cognizant of the fact that a limited stack might exist at the time that the **put** procedure is invoked. However, *STREAMS* also provides the **service** procedure as a way to defer processing to a '`!in_irq()`' context.

4.1.2 Service Procedure

Each queue in module or driver queue pair can also have a **service** procedure associated with it.

A queue's **service** procedure is specified by the `qi_srvp` member of the `qinit(9)` structure associated with the `queue(9)`. If a queue does not have a **service** procedure, the associated `qi_srvp` member is set to `NULL`. If the queue has a **service** procedure, the associated `qi_srvp` member points to the **service** procedure function. As with **put** procedures, in general, the read- and write-side queues of a module or driver have different `qinit(9)` structure associated with them as there are normally differences between the upstream and downstream message processing; however, it is possible for read- and write-side queues to share the same `qinit(9)` structure.

A queue's **service** procedure is never invoked directly by an adjacent module or driver. Adjacent modules or drivers invoke a queue's **put** procedure. The **put** procedure can then defer processing to the **service** procedure in a number of ways. The most direct way that a **put** procedure can invoke a **service** procedure for a message is to place that message on the message queue using `putq(9)`. Once the message is placed on the message queue in this

⁴ Because the *Interrupt Service Routine (ISR)* stack is particularly limited, `put(9)` should not be called from '`in_irq()`' context under *Linux*, execution of `put(9)` should be deferred by the ISR, either with an immediate bottom half procedure (i.e., software interrupt), or by placing messages on the driver queue and processing from the queue's **service** procedure: either of which run with a full kernel stack instead of an interrupt stack.

manenr, the `put` procedure can return, freeing the associated stack frame. Also, placing a message on the message queue with `putq(9)` will normally result in the queue's `service` procedure being *scheduled* for later execution by the *STREAMS* scheduler.

Note that the *STREAMS* scheduler is separate and distinct from the *Linux* scheduler. The *Linux* scheduler is responsible for scheduling tasks, whereas the *STREAMS* scheduler is only responsible for scheduling the execution of queue `service` procedures (and a few other deferrable *STREAMS* housekeeping chores). The *STREAMS* scheduler executes pending queue `service` procedures on a *First-Come-First-Served (FCFS)* basis. When a queue's `service` procedure is scheduled, its `queue(9)` structure is linked onto the tail of the list of queues awaiting `service` procedure execution for the *STREAMS* scheduler. When the *STREAMS* scheduler runs queues, each queue on the list is unlinked, starting at the head of the list, and its `service` procedure executed.

To provide reponsive scheduling of `service` procedures without necessarily requiring a task switch (to the *STREAMS* kernel thread), the *STREAMS* scheduler is invoked and queue `service` procedures executed within user context before returning to user level from any *STREAMS* system call.

Processing of messages within a queue `service` procedure is performed by taking messages off of the message queue and processing them in order. Because messages are queued on the message queue with consideration to the priority class of the message, messages of higher priority are processed by the `service` procedure first. However, providing that no other condition impedes further processing of messages (e.g. flow control, inability to obtain a message block), `service` procedures process all of the messages on the message queue available to them and then return. Because `service` procedures are invoked by the *STREAMS* scheduler on a *FCFS* basis, a priority message on a queue does not increase the scheduling priority of a queue's `service` procedure with respect to other queue `service` procedures: it only affects the priority of processing one message on message queue with respect to other messages on the queue. As a result, higher priority messages will experience a shorter processing latency than lower priority messages.

In general, because drivers run at a software priority higher than the *STREAMS* scheduler, drivers calling `put(9)` can cause multiple messages to be queued for service before the `service` procedure runs. On the other hand, because the *STREAMS* scheduler is always invoked before return to user level at the end of a system call, it is unlikely that the *Stream head* calling `put(9)` will result in multiple messages being accumulated before the corresponding `service` procedure runs.

4.1.3 Put and Service Procedure Summary

Processing of messages can be divided between `put` and `service` procedures to meet the requirements for *STREAMS* processing, and to meet the demands of the module or driver. Some message types might be processed entirely within the `put` procedure. Others might be processed only with the `service` procedure. A third class of messages might have processing split between `put` and `service` procedures. Processing of upstream and downstream messages can be independent, giving consideration to the needs of each message flow. The mechanism allows a flexible arrangement for the module and driver writer.

`put` and `service` procedures are addressed in more detail under [Chapter 7 \[Modules and Drivers\]](#), page 95. Design guildlines for `put` and `service` processing are given in [\[undefined\]](#) [undefined], page undefined, [\[undefined\]](#) [undefined], page undefined, and [\[undefined\]](#) [undefined], page undefined.

4.2 Asynchronous Example

5 Messages

5.1 Messages Overview

All communications between the *Stream head*, modules and drivers within the *STREAMS* framework is based on message passing. Control and data information is passed along the *Stream* as opposed to direct function calls between modules. Adjacent modules and driver are invoked by passing pointers to messages to the target queue's **put** procedure. This permits processing to be deferred (i.e. to a **service** procedure) and to be subjected to flow control and scheduling within the *STREAMS* framework.

At the *Stream head*, conversion between functional call based systems calls and the message oriented *STREAMS* framework is performed. Some system calls retrieve upstream messages or information about upstream messages at the *Stream head* queue pair, others create messages and pass them downstream from the *Stream head*.

At the *Stream end* (driver), conversion between device or pseudo-device actions and events and *STREAMS* messages is performed in a similar manner to that at the *Stream head*. Downstream control messages are consumed converted into corresponding device actions, device events generate appropriate control messages and the driver sends these upstream. Downstream messages containing data are transferred to the device, and data received from the device is converted to upstream data messages.

Within a linear segment from *Stream head* to *Stream end*, messages are modified, created, destroyed and passed along the *Stream* as required by each module in the *Stream*.

Messages consist of a 3-tuple of a message block structure (**msgb(9)**), a data block structure (**datab(9)**) and a data buffer. The message block structure is used to provide an instance of a reference to a data block and pointers into the data buffer. The data block structure is used to provide information about the data buffer, such as message type, separate from the data contained in the buffer. Messages are normally passed between *STREAMS* modules, drivers and the *Stream head* using utilities that invoke the target module's **put** procedure, such as **put(9)**, **putnext(9)**, **qreply(9)**. Messages travel along a *Stream* with successive invocations of each driver, module and *Stream head*'s **put** procedure.

5.1.1 Message Types

Each data block (**datab(9)**) is assigned a message type. The message type discriminates the use of the message by drivers, modules and the *Stream head*. Message types are defined in '**sys/stream.h**'. Most of the message types may be assigned by a module or driver when it generates a message, and the message type can be modified as a part of message processing. The *Stream head* uses a wider set of message types to perform its function of converting the functional interface to the user process into the messaging interface used by *STREAMS* modules and drivers.

Most of the defined message types are solely for use within the *STREAMS* framework. A more limited set of message types (**M_PROTO**, **M_PCPROTO** and **M_DATA**) can be used to pass control and data information to and from the user process via the *Stream head*. These message type can be generated and consumed using the **read(2)**, **write(2)**, **getmsg(2)**,

`getpmsg(2s)`, `putmsg(2)`, `putpmsg(2s)` system calls and some `streamio(7)` *STREAMS* `ioctl(2)`.

Below the message types are classified by queueing priority, direction of normal travel (downstream or upstream), and briefly described:

5.1.1.1 Ordinary Messages

Ordinary Messages (also called normal messages) are listed in the table below. Messages with a ‘D’ beside them can normally travel in the *downstream* direction; with a ‘U’, *upstream*. Messages with an ‘H’ beside them can be generated by the *Stream head*; an ‘M’, a module; an ‘E’, the *Stream end* or driver. Messages with an ‘h’ beside them are consumed and interpreted by the *Stream head*; an ‘m’, interpreted by a module; an ‘e’, consumed and interpreted by the *Stream end* or driver.

The following message types are defined by *SVR 4.2*:

M_DATA	D	U	HME	hme	User data message for I/O system calls
M_PROTO	D	U	HME	hme	Protocol control information
M_BREAK	D	-	ME	me	Request to a <i>Stream</i> driver to send a "break"
M_PASSFP	-	U	H	h	File pointer passing message ¹
M_SIG	-	U	ME	h	Signal sent from a module/driver to a user
M_DELAY	D	-	ME	me	Request a real-time delay on output
M_CTL	D	U	ME	me	Control/status request used for inter-module communication
M_IOCTL	D	-	H	me	Control/status request generated by a <i>Stream head</i>
M_SETOPTS	-	U	ME	h	Set options at the <i>Stream head</i> , sent upstream
M_RSE	D	U	ME	me	Reserved for internal use

The following message types are *not* defined by *SVR 4.2* and are *Linux Fast-STREAMS* specific, or are specific to another *SVR 4.2*-based implementation:

M_EVENT	
M_TRAIL	
M_BACKWASH	AIX specific message for driver direct I/O.

Ordinary messages are described in detail throughout this chapter and in [Appendix B \[Message Types\]](#), page 117.

5.1.1.2 High Priority Messages

High Priority Messages messages are listed in the table below. Messages with a ‘D’ beside them can normally travel in the *downstream* direction; with a ‘U’, *upstream*. Messages with an ‘H’ beside them can be generated by the *Stream head*; an ‘M’, a module; an ‘E’, the *Stream end* or driver. Messages with an ‘h’ beside them are consumed and interpreted by the *Stream head*; an ‘m’, interpreted by a module; an ‘e’, consumed and interpreted by the *Stream end* or driver.

The following message types are defined by *SVR 4.2*:

¹ M_PASSFP is never passed on the *Stream* but is placed on one *Stream head* directly by the opposite *Stream head* of a *STREAMS*-based pipe.

M_IOCACK	-	U	ME	h	Positive <code>ioctl(2)</code> acknowledgement
M_IOCNAK	-	U	ME	h	Negative <code>ioctl(2)</code> acknowledgement
M_PCPROTO	D	U	HME	hme	Protocol control information
M_PCSIG	-	U	ME	h	Signal sent from a module/driver to a user
M_READ	D	-	H	me	Read notification, sent downstream
M_FLUSH	D	U	HME	hme	Flush module queue
M_STOP	D	-	ME	me	Suspend output
M_START	D	-	ME	me	Restart stopped device output
M_HANGUP	-	U	ME	h	Set a <i>Stream head</i> hangup condition, sent upstream
M_ERROR	-	U	ME	h	Report downstream error condition, sent upstream
M_COPYIN	-	U	ME	h	Copy in data for transparent ² <code>ioctl</code> s, sent upstream
M_COPYOUT	-	U	ME	h	Copy out data for transparent ³ <code>ioctl</code> s, sent upstream
M_IOCDATA	D	-	H	me	Data for transparent ⁴ <code>ioctl</code> s, sent downstream
M_PCRSE	D	U	ME	hme	Reserved for internal use
M_STOPI	D	-	ME	me	Suspend input
M_STARTI	D	-	ME	me	Restart stopped device input

The following message types are *not* defined by *SVR 4.2* and are *Linux Fast-STREAMS* specific, or are specific to another *SVR 4.2*-based implementation:

M_PCCTL	D	U	ME	me	Same as M_CTL, but high priority.
M_PCSETOPTS	-	U	ME	h	Same as M_SETOPTS, but high priority.
M_PCEVENT					Same as M_EVENT, but high priority.
M_UNHANGUP	-	U	ME	h	Reverses a previous M_HANGUP message.
M_NOTIFY					
M_HPDATA	D	U	HME	hme	Same as M_DATA, but high priority.
M_LETSPLAY					AIX specific message for driver direct I/O.
M_DONTPLAY					AIX specific message for driver direct I/O.
M_BACKDONE					AIX specific message for driver direct I/O.
M_PCTTY					

High Priority messages are described in detail throughout this chapter and in [Appendix B \[Message Types\]](#), page 117.

5.1.2 Expedited Data

5.2 Message Structure

STREAMS messages consist of a chain of one or more message blocks. A message block is a triplet of a `msgb(9)` structure, a `datab(9)` structure, and a variable length data buffer. A message block (`msgb(9)` structure) is an instance of a reference to the data contained in the data buffer. Many message block structures can refer to a data block and data buffer.

² Transparent `ioctl`s support applications developed prior to the introduction of *STREAMS*.

³ Ibid.

⁴ Ibid.

A data block (`datab(9)` structure) contains information not contained in the data buffer, but directly associated with the data buffer (e.g., the size of the data buffer). One and only one data block is normally associated with each data buffer. Data buffers can be internal to the message block, data block, data buffer triplet, automatically allocated using `kmem_alloc(9)`, or allocated by the module or driver and associated with a data block (i.e., using `esballoc(9)`).

The `msgb(9)` structure is defined in ‘`sys/stream.h`’ and has the following format and members:

```
typedef struct msgb {
    struct msgb *b_next;           /* next msgb on queue */
    struct msgb *b_prev;           /* prev msgb on queue */
    struct msgb *b_cont;           /* next msgb in message */
    unsigned char *b_rptr;         /* rd pointer into datab */
    unsigned char *b_wptr;         /* wr pointer into datab */
    struct datab *b_datab;         /* pointer to datab */
    unsigned char b_band;          /* band of this message */
    unsigned char b_pad1;          /* padding */
    unsigned short b_flag;         /* message flags */
    long b_pad2;                   /* padding */
} mblk_t;
```

The members of the `msgb(9)` structure are described as follows:

<code>b_next</code>	points to the next message block on a message queue;
<code>b_prev</code>	points to the previous message block on a message queue;
<code>b_cont</code>	points to the next message block in the same message chain;
<code>b_rptr</code>	points to the beginning of the data (the point from which to read);
<code>b_wptr</code>	Points to the end of the data (the point from which to write);
<code>b_datab</code>	points to the associated datablock (<code>datab(9)</code>);
<code>b_band</code>	indicates the priority band;
<code>b_pad1</code>	provides padding; and
<code>b_flag</code>	holds flags for this message block. Flags are normally set only on the first block of a message. Valid flags are discussed below.
<code>b_pad2</code>	Reserved. ⁵

The `b_band` member determines the priority band of the message. This member determines the queueing priority (placement) in a message queue when the message type is an ordinary message type. High priority message types are always queued ahead of ordinary message types, and the `b_band` member is always set to ‘0’ whenever a high priority message is queued by a *STREAMS* utility function. When `allocb(9)` or `esballoc(9)` are used to allocate a message block, the `b_band` member is initially set to ‘0’. This member may be modified by a module or driver.

Note that in *System V Release 4.0*, certain data structures fundamental to the kernel (for example, device numbers, user IDs) were enlarged to enable them to

⁵ Note that *Linux Fast-STREAMS* does not include the `b_pad2` member to reduce the size of the triplet and provide more room for a cache-aligned internal data buffer.

hold more information. This feature was referred to as Expanded Fundamental Types (EFT). Since some of this information was passed in *STREAMS* messages, there was a binary compatibility issue for pre-*System V Release 4* drivers and modules. `#ifdef`'s were added to the kernel to provide a transition period for these drivers and modules to be recompiled, and to allow it to be built to use the pre-*System V Release 4* short data types or the *System V Release 4* long data types. Support for short data types will be dropped in some future releases.⁶

The values that can be used in *b_flag* are exposed when `'sys/stream.h'` is included:

```
#define MSGMARK      (1<<0) /* last byte of message is marked */
#define MSGNOLOOP    (1<<1) /* don't loop message at stream head */
#define MSGDELIM     (1<<2) /* message is delimited */
#define MSGNOGET     (1<<3) /* UnixWare/Solaris/Mac OT/ UXP/V getq does not
                           return message */
#define MSGATTEN     (1<<4) /* UXP/V attention to on read side */
#define MSGMARKNEXT  (1<<4) /* Solaris */
#define MSGLOG       (1<<4) /* UnixWare */
#define MSGNOTMARKNEXT (1<<5) /* Solaris */
#define MSGCOMPRESS  (1<<8) /* OSF: compress like messages as space allows */
#define MSGNOTIFY    (1<<9) /* OSF: notify when message consumed */
```

The following flags are defined by *SVR 4.2*:

MSGMARK	last byte of message is marked
MSGNOLOOP	don't loop message at stream head
MSGDELIM	message is delimited

The following flags are *not* defined by *SVR 4.2* and are *Linux Fast-STREAMS* specific, or are specific to another *SVR 4.2*-based implementation:

MSGNOGET	UnixWare/Solaris/Mac OT/ UXP/V getq does not return message
MSGATTEN	UXP/V attention to on read side
MSGMARKNEXT	Solaris
MSGLOG	UnixWare
MSGNOTMARKNEXT	Solaris
MSGCOMPRESS	OSF: compress like messages as space allows
MSGNOTIFY	OSF: notify when message consumed

⁶ System V Release 4 Programmer's Guide: STREAMS.

```

typedef struct free_rtn {
    void (*free_func) (caddr_t);
    caddr_t free_arg;
} frtn_t;

typedef struct datab {
    union {
        struct datab *freep;
        struct free_rtn *frtnp;
    } db_f;
    unsigned char *db_base;
    unsigned char *db_lim;
    unsigned char db_ref;
    unsigned char db_type;
    unsigned char db_class;
    unsigned char db_pad;
    unsigned int db_size;

#if 0
    unsigned char db_cache[DB_CACHESIZE];
#endif
#if 0
    unsigned char *db_msgaddr;
    long db_filler;
#endif
    /* Linux Fast-STREAMS specific members */
    atomic_t db_users;
} dblk_t;

#define db_freep db_f.freep
#define db_frtnp db_f.frtnp

```

The following members are defined by *SVR 4.2*:

<i>db_freep</i>	pointer to an external data buffer to be freed;
<i>db_frtnp</i>	pointer to an routine to free an extended buffer;
<i>db_base</i>	base of the buffer (first usable byte);
<i>db_lim</i>	limit of the buffer (last usable byte plus 1);
<i>db_ref</i>	numer of references to this data block by message blocks;
<i>db_type</i>	the data block type (i.e., <i>STREAMS</i> message type);
<i>db_class</i>	the class of the message (normal or high priority);
<i>db_iswhat</i>	another name for <i>db_class</i> ;
<i>db_pad</i>	padding;
<i>db_filler2</i>	another name for <i>db_pad</i> ;
<i>db_size</i>	size of the buffer;
<i>db_cache</i>	SVR 3.1 internal buffer; ⁷

⁷ This is an old SVR 3.1 member that was used to contain the internal data buffer. It is not longer at this location and this member is not present in *Linux Fast-STREAMS*.

db_msgaddr pointer to `msgb(9)` structure allocated with this data block in a 3-tuple;⁸

db_filler filler; and,⁹

The following members are *not* defined by *SVR 4.2* and are *Linux Fast-STREAMS* specific:

db_users same as *db_ref* but atomic.

5.2.1 Message Linkage

The message block (`msgb(9)` structure) provides an instance of a reference to the data buffer associated with the message block. Multiple message blocks can be chained together (with *b_cont* pointers) into a composite message. When multiple messages blocks are chained the type of the first message block (its *db_type*) determines the type of the overall message. For example, a message consisting of an `M_IOCTL` message block followed by an `M_DATA` message block is considered to be an `M_IOCTL` message. Other message block members of the first message block, such as *b_band*, also apply to the entire message. The initial message block of a message block chain can be queued onto a message queue (with the *b_next* and *b_prev* pointers). The chaining of message blocks into messages using the *b_cont* pointer, and linkage onto message queues using the *b_next* and *b_prev* pointers, are illustrated in Figure 5.1.

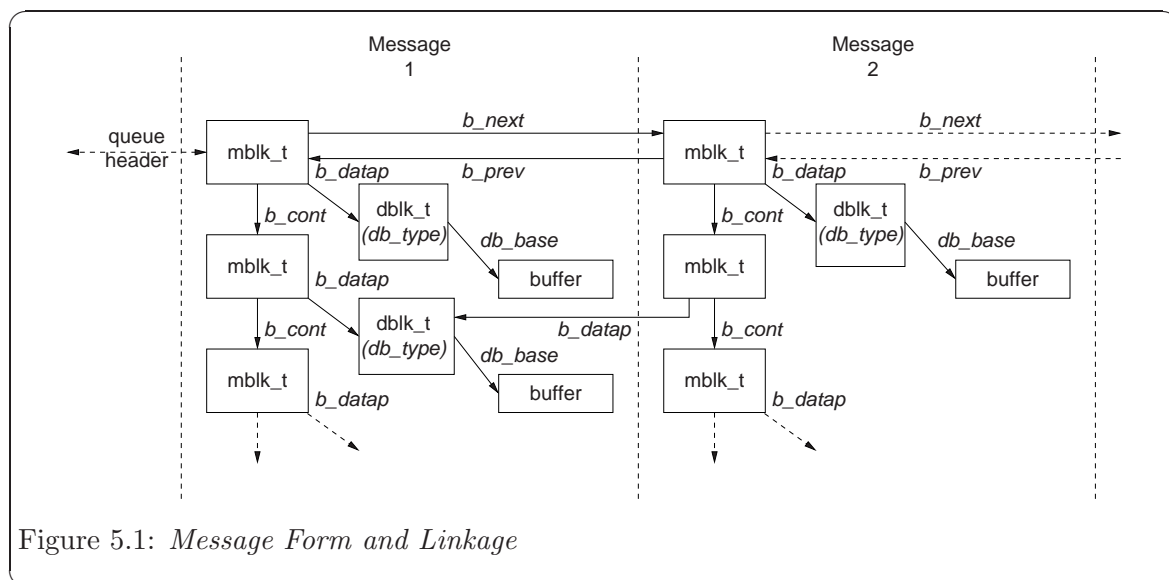


Figure 5.1: *Message Form and Linkage*

A message can occur stand-alone (that is, it is not present on any message queue as it is in a module or driver's `put` procedure) or can be queued on a message queue awaiting processing by the queue's `service` procedure. The *b_next* and *b_prev* pointers are not significant for

⁸ This member is used by some implementations to locate the initial `msgb(9)` structure allocated with this data block as a 3-tuple. *Linux Fast-STREAMS* calculates this address from the address of the data block itself and discards this member to reduce the overall size of the 3-tuple and to increase the cache-aligned size of the internal data buffer.

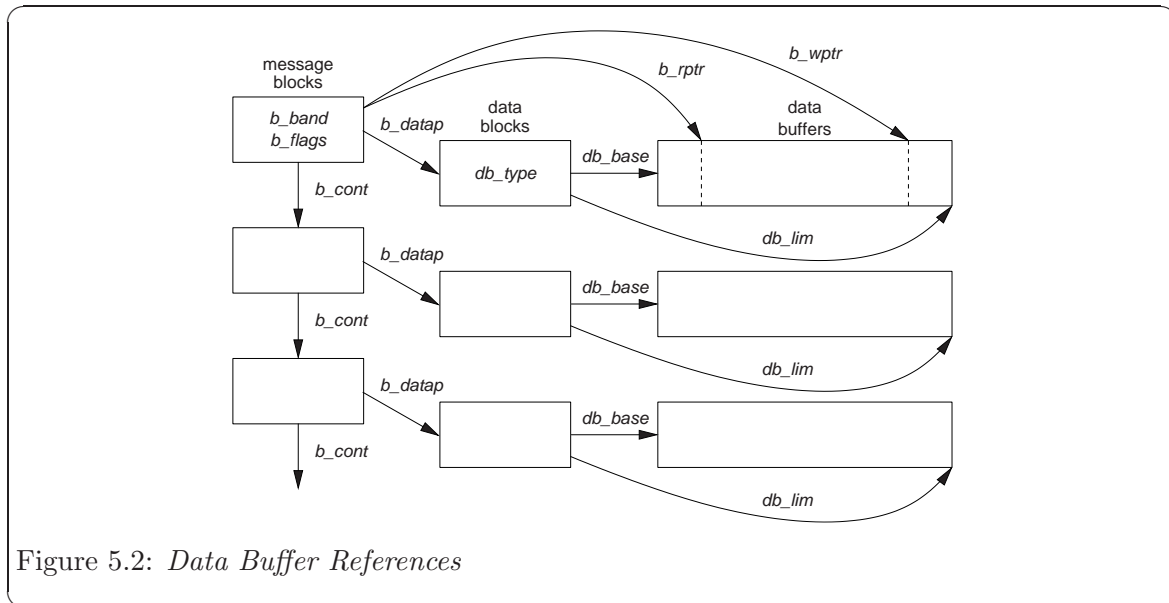
⁹ *Linux Fast-STREAMS* discards this field to reduce the overall size of the structure and to increase the cache-aligned size of the internal data buffer.

a stand-alone message and are initialized to `NULL` by *STREAMS* when the message is not queued on a message queue.

A message block is an instance of a reference to a data block (and therefore data buffer). Multiple message block can refer to the same data block. This is illustrated in Figure 5.1. In the figure, the second message block of ‘**Message 1**’ shares a data block with the second message block of ‘**Message 2**’. Message blocks that share data blocks result from use of the `dupb(9)` and `dupmsg(9)` *STREAMS* utilities. The first of these utilities, `dupb(9)`, will duplicate a message block, obtaining a new reference to the data block. The `db_ref` member of the associated data block will be increased by one to indicate the number of message blocks that refer to this data block. The second of these utilities, `dupmsg(9)` duplicate all of the message blocks in a message, following the `b_cont` pointers, resulting in a duplicated message.

Duplication of message blocks provides an excellent way of obtaining a new reference to a data buffer without the overhead of copying each byte of the buffer. A common use of duplication is to obtain a duplicate of a message to be held for retransmission, while another duplicate is passed to the next module for transmission.

Despite the advantages of duplication, copying a message block or message chain is also possible with the `copyb(9)` and `copymsg(9)` *STREAMS* utilities. These utilities copy the message block, data block, and data buffer for one message block (`copyb(9)`) or each message block in a chain (`copymsg(9)`).



Being a reference to a data buffer, the message block has two pointer into the data buffer that define the range of data used by the reference. The `b_rptr` indicates the beginning of the range of data in the data buffer, and represents the position at which a module or driver would begin reading data; the `b_wptr`, the end of the range of data, where a module or driver would begin writing data. The data block, on the other hand, has two pointers representing the absolute limits of the data buffer. The `db_base` indicates the beginning of

the data buffer; *db_lim*, the end. This relationship between pointers into the data buffer is illustrated in [Figure 5.2](#).

STREAMS provides a library of utility functions used to manipulate message blocks, data blocks and data buffers. The members of a message block or data block should not be manipulated directly by the module or driver write: an appropriate *STREAMS* message utility should be used instead. See [Appendix C \[Utilities\]](#), page 119.

5.2.2 Sending and Receiving Messages

As shown in the message lists of [Section 5.1 \[Messages Overview\]](#), page 65, a large subset of the available message types can be generated and consumed by modules and drivers. Another subset, are dedicated to generation and consumption by the *Stream head*.

Message types that are dedicated for passing control and data information between the *Stream* and a user level process are the `M_PROTO`, `M_PCPROTO`, and `M_DATA` messages.¹⁰ *STREAMS*-specific system calls are provided to user level processes so that they may exchange `M_PROTO`, `M_PCPROTO` and `M_DATA` message with a *Stream*. This permits a user level process to interact with the *Stream* in a similar fashion as a module on the *Stream*, allowing user level processes to also present a service interface.¹¹

In general, all system calls interact directly (by subroutine interface) with the *Stream head*. An exception is the `open(2)` and `close(2)` system calls which directly invoke a subroutine call to the module or driver *qi_qopen* and *qi_qclose* procedures. All other system calls call subroutines provided by the *Stream header* that can result in the generation and transmission of a message on the *Stream* from the *Stream head*, or consumption of a message at the *Stream head*.

The traditional `write(2)` system call is capable of directly generating `M_DATA` messages and having them passed downstream. The traditional `read(2)` system call can collect `M_DATA` messages (and in some read modes, `M_PROTO` and `M_PCPROTO` messages) that have arrived at the *Stream head*. These system calls provide a backward compatible interface for character device drivers implemented under *STREAMS*.¹²

The *STREAMS*-specific `putmsg(2)`, `putpmsg(2s)` system calls provide the user level process with the ability to directly generate `M_PROTO`, `M_PCPROTO` or `M_DATA` messages and have send downstream on the *Stream* from the *Stream head*. `getmsg(2)`, `getpmsg(2s)` system calls provide the ability to collect `M_PROTO`, `M_PCPROTO` and `M_DATA` messages from the *Stream head*. These system calls are superior to the `write(2)` and `read(2)` system calls in that they provide a finer control over the composition of the generated message, and more information concerning the composition of a consumed message. Whereas, `write(2)` and `read(2)` pass only one buffer from the user, `putmsg(2)`, `putpmsg(2s)`, `getmsg(2)`, `getpmsg(2s)` provide two buffers: one for the control part of the message to transfer `M_PROTO` or `M_PCPROTO`

¹⁰ Some *SVR 4.2*-based implementations also provide the `M_HPDATA` message for passing high priority data in the same fashion as `M_DATA` messages.

¹¹ For a complete applications framework based on *STREAMS* and service interfaces, see the [ADAPTIVE Communications Environment \(ACE\)](#) communications framework

¹² One example of backwards compatibility to a character device driver implemented under *STREAMS* is the *STREAM* implementation of terminal and pseudo-terminal devices.

message blocks with preservation of boundaries; another for the data part, to transfer `M_DATA` messages blocks – all in a single call. Also, data transfer with `write(2)` and `read(2)` are by nature byte-stream oriented, whereas, control and data transfer with `putmsg(2)` and `getmsg(2)` are by nature message oriented. `write(2)` and `read(2)` provide no mechanism for assigning priority to messages generated or indicating the priority of messages to be received: `putpmsg(2s)` and `getpmsg(2s)` provide the ability to specify criteria for the band (*b.band*) of the generated or consumed message.

5.2.2.1 putmsg(2)

`putmsg(2)` provides the ability for a user level process to generate `M_PROTO`, `M_PCPROTO` and `M_DATA` messages and have them send downstream on a *Stream*. The user specifies a control part of the message that is used to fill the `M_PROTO` or `M_PCPROTO` message block in the resulting message, and a data part of the message that is used to fill the `M_DATA` message block in the resulting message.

The prototype for the `putmsg(2)` system call is exposed by including the ‘`sys/stropts.h`’ system header file. The prototype for the `putmsg(2)` system call is as follows:

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

Where the arguments are interpreted as follows:

- | | |
|----------------|--|
| <i>fildes</i> | specifies the <i>Stream</i> upon which to generate messages and is a file descriptor that was returned by the corresponding call to <code>open(2)</code> or <code>pipe(2)</code> that created the <i>Stream</i> . |
| <i>ctlptr</i> | is a pointer to a read-only <code>strbuf(5)</code> structure that is used to specify the control part of the message. |
| <i>dataptr</i> | is a pointer to a read-only <code>strbuf(5)</code> structure that is used to specify the data part of the message. |
| <i>flags</i> | specifies whether the control part of the message is to be of type <code>M_PROTO</code> or of type <code>M_PCPROTO</code> . It can have values ‘0’ (specifying that an <code>M_PROTO</code> message be generated) or ‘ <code>RS_HIPRI</code> ’ (specifying that an <code>M_PCPROTO</code> message be generated). |

The *ctlptr* and *dataptr* point to a `strbuf(5)` structure that is used to specify the control and data parts of the message. The `strbuf(5)` structure has the format and members as follows:

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

The members of the `strbuf(5)` structure are interpreted by `putmsg(2)` as follows:

- | | |
|---------------|---|
| <i>maxlen</i> | specifies the maximum length of the buffer and is ignored by <code>putmsg(2)</code> ; |
|---------------|---|

len specifies the length of the data for transfer in the control or data part of the message; and,

buf specifies the location of the data buffer containing the data for transfer in the control or data part of the message.

If *ctlptr* is set to 'NULL' on call, or the *len* member of the `strbuf(5)` structure pointed to by *ctlptr* is set to '-1', then no control part (`M_PROTO` or `M_PCPROTO` message block) will be placed in the resulting message.

If *dataptr* is set to 'NULL' on call, or the *len* member of the `strbuf(5)` structure pointed to by *dataptr* is set to '-1', then no data part (`M_DATA` message block) will be placed in the resulting message.

For additional details, see the `putmsg(2)` or `putmsg(2p)` reference page.

5.2.2.2 `getmsg(2)`

`getmsg(2)` provides the ability for a user level process to retrieve `M_PROTO`, `M_PCPROTO` and `M_DATA` messages that have arrived at the *Stream head*. The user specifies an area into which to receive any control part of the message (from `M_PROTO` or `M_PCPROTO` message blocks in the message), and an area into which to receive any data part of the message (from `M_DATA` message blocks in the message).

The prototype for the `getmsg(2)` system call is exposed by including the '`sys/stropts.h`' system header file. The prototype for the `getmsg(2)` system call is as follows:

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

Where the arguments are interpreted as follows:

fildes specifies the *Stream* upon which to generate messages and is a file descriptor that was returned by the corresponding call to `open(2)` or `pipe(2)` that created the *Stream*.

ctlptr is a pointer to an `strbuf(5)` structure that is used to specify the area to accept the control part of the message.

dataptr is a pointer to an `strbuf(5)` structure that is used to specify the area to accept the data part of the message.

flagsp is a pointer to an integer flags word that is used both to specify the criteria for the type of message to be retrieved, on call, as well as indicating the type of the message retrieved, on return.

On call, the integer pointed to by *flagsp* can contain '0' indicating that the first available message is to be retrieved regardless of priority; or, '`RS_HIPRI`', indicating that only the first high priority message is to be retrieved and no low priority message. On successful return, the integer pointed to by *flagsp* will contain '0' to indicate that the message retrieved was an ordinary message (`M_PROTO` or just `M_DATA`), or '`RS_HIPRI`' to indicate that the message retrieved was of high priority (`M_PCPROTO` or just `M_HPDATA`).

The members of the `strbuf(5)` structure are interpreted by `getmsg(2)` as follows:

<i>maxlen</i>	specifies the maximum length of the buffer into which the message part is to be written;
<i>len</i>	ignored by <code>getmsg(2)</code> on call, but set on return to indicate the length of the data that was actually written to the buffer by <code>getmsg(2)</code> ; and,
<i>buf</i>	specifies the location of the data buffer to contain the data retrieved for the control or data part of the message.

If *ctlptr* or *dataptr* are 'NULL' on call, or the *maxlen* field of the corresponding `strbuf(5)` structure is set to '-1', then `getmsg(2)` will not retrieve the corresponding control or data part of the message.

For additional details, see the `getmsg(2)` or `getmsg(2p)` reference page.

5.2.2.3 putpmsg(2s)

`putpmsg(2s)` is similar to `putmsg(2)`, but provides the additional ability to specify the queue priority band (*b.band*) of the resulting message. The prototype for the `putpmsg(2s)` system call is exposed by including the '`sys/stropts.h`' system header file. The prototype for the `putpmsg(2s)` system call is as follows:

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

The arguments to `putpmsg(2s)` are interpreted the same as those for `putmsg(2)` as described in [Section 5.2.2.1 \[putmsg\(2\)\]](#), [page 74](#) with the exception of the *band* and *flags* arguments.

The *band* argument provides a band number to be placed in the *b.band* member of the first message block of the resulting message. *band* can only be non-zero if the message to be generated is a normal message.

The *flags* argument is interpreted differently by `putpmsg(2s)`: it can have values 'MSG_BAND' or 'MSG_HIPRI', but these are equivalent to the '0' and 'RS_HIPRI' flags for `putmsg(2)`.

Under *Linux Fast-STREAMS*, `putmsg(2)` is implemented as a library call to `putpmsg(2s)`. This is possible because the call:

```
putmsg(fildes, ctlptr, dataptr, flags);
```

is equivalent to:

```
putpmsg(fildes, ctlptr, dataptr, 0, flags);
```

For additional details, see the `putpmsg(2s)` or `putpmsg(2p)` reference page.

5.2.2.4 getpmsg(2s)

`getpmsg(2s)` is similar to `getmsg(2)`, but provides the additional ability to specify the queue priority band (*b.band*) of the retrieved message. The prototype for the `getpmsg(2s)` system call is exposed by including the '`sys/stropts.h`' system header file. The prototype for the `getpmsg(2s)` system call is as follows:


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

The arguments to `getpmsg(2s)` are interpreted the same as those for `getmsg(2)` as described in [Section 5.2.2.2 \[getmsg\(2\)\]](#), page 75, with the exception of the *bandp* and *flagsp* arguments.

The *bandp* argument points to a band number on call that specifies a criteria for use with selecting the band of the retrieved message and returns the band number of the retrieved message upon successful return. The integer pointed to by *bandp* can take on values as follows:

MSG_ANY Only specified on call. Specifies that the first available message is to be retrieved, regardless of priority or band.

MSG_BAND On call, specifies that an ordinary message of message band *bandp* or greater is to be retrieved. On return, indicates that an ordinary message was retrieved of the band returned in *bandp*.

MSG_HIPRI On call, specifies that a high priority message is to be retrieved. On return, indicates that a high priority message was retrieved.

On call, *bandp* is ignored unless *flagsp* specifies ‘MSG_BAND’. When ‘MSG_BAND’ is specified, *bandp* specifies the minimum band number of the message to be retrieved. On return, *bandp* indicates the band number (*b_band*) of the retrieved message, or ‘0’ if the retrieved message was a high priority message.

Under *Linux Fast-STREAMS*, `getmsg(2)` is implemented as a library call to `getpmsg(2s)`. This is possible because the calls:

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

```
int flags = RS_HIPRI;
getmsg(fildes, ctlptr, dataptr, &flags);
```

are equivalent to:

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

```
int band = 0;
int flags = MSG_HIPRI;
getpmsg(fildes, ctlptr, dataptr, &band, &flags);
```

For additional details, see the `getpmsg(2s)` or `getpmsg(2p)` reference page.

5.2.3 Control of Stream Head Processing

Stream head message processing can be controlled by the user level process, or by a module or driver within the *Stream*.

Modules and drivers can control *Stream head* processing using the `M_SETOPTS` message. At any time, a module or driver can issue an `M_SETOPTS` message upstream. The `M_SETOPTS`

contains a `stroptions(9)` structure (see [Appendix A \[Data Structures\]](#), page 115) specifying which *Stream head* characteristics to alter in the read-side queue of the *Stream head* (including `q_hiwat`, `qi_lowat`, `q_minpsz` and `q_maxpsz`), however, of interest to the current discussion are the read and write options associated with the *Stream head*.

User level processes can also alter the read and write options associated with the *Stream head*. User level processes use the `I_SRDOPT(7)`, `I_GRDOPT(7)`, `I_SWROPT(7)` and `I_GWROPT(7)` `ioctl(2)` commands to achieve the same purpose as the `M_SETOPTS` message used by modules and drivers.

5.2.3.1 Read Options

Read options are altered by a user level process using the `I_SRDOPT(7)` and `I_GRDOPT(7)` `ioctl(2)` commands; or altered by a module or driver using the `SO_READOPT` flag and `so_readopt` member of the `stroptions(9)` data structure contained in an `M_SETOPTS` message passed upstream.

Two flags, each selected from two sets of flags, can be set in this manner. The two sets of flags are as follows:

5.2.3.2 Read Mode

The read mode affects how the `read(2)` and `readv(2)` system calls treat message boundaries. One read mode can be selected from the following modes:

- | | |
|--------------|---|
| RNORM | byte-stream mode. This is the default read mode. This is the normal byte-stream mode where message boundaries are ignored. <code>read(2)</code> and <code>readv(2)</code> return data until the read count has been satisfied or a zero length message is received. |
| RMSGD | message non-discard mode. The <code>read(2)</code> and <code>readv(2)</code> system calls will return when either the count is satisfied, a zero length message is received, or a message boundary is encountered. If there is any data left in a message after the read count has been satisfied, the message is placed back on the <i>Stream head</i> read queue. The data will be read on a subsequent <code>read(2)</code> or <code>readv(2)</code> call. |
| RMSGN | message discard mode. Similar to RMSGD mode, above, but data that remains in a message after the read count has been satisfied is discarded. |
| RFILL | message fill mode. Similar to RNORM but requests that the <i>Stream head</i> fill a buffer completely before returning to the application. This is used in conjunction with a cooperating module and <code>M_READ</code> messages. ¹³ |

5.2.3.3 Read Protocol

The read protocol affects how `read(2)` and `readv(2)` system calls treat the control part of a message. One read protocol can be selected from the following protocols:¹⁴

¹³ The **RFILL** option is not defined by *SVR 4.2*, but is defined by some implementations based on *SVR 4.2*.

¹⁴ Note that earlier releases, such as *UNIX System V Release 3.0*, did not support read protocols. Under these earlier implementations, the read protocol was always **RPROTNORM**.

RPROTNORM

fail read when control part present. Fail `read(2)` with `[EBADMSG]` if a message containing a control part is at the front of the *Stream head* read queue. Otherwise, the message will read as normal. This is the default setting for new *Stream heads*.¹⁵

RPROTDAT deliver control part of a message as data. The control part of a message is prepended to the data part and delivered.¹⁶

RPROTDIS discard control part of message, delivering only any data part. The control part of the message is discarded and the data part is processed.¹⁷

RPROTCOMPRESS

compress like data.¹⁸

Note that, although all modes terminate the read on a zero-length message, *POSIX* requires that zero only be returned from `read(2)` when the requested length is zero or an end of file (`M_HANGUP`) has occurred. Therefore, *Linux Fast-STREAMS* only returns on a zero-length message if some data has been read already.

5.2.3.4 Write Options

No mechanism is provided to permit a `write(2)` system call to generate either a `M_PROTO` or `M_PCPROTO` message. The `write(2)` system call will only generate one or more `M_DATA` messages.

Write options are altered by a user level process using the `I_SWROPT(7)` and `I_GWROPT(7)` `ioctl(2)` commands. It is not possible for a module or driver to affect these options with the `M_SETOPTS` message.

SNDZERO Permits the sending of a zero-length message downstream when a `write(2)` of zero length is issued. Without this option being set, `write(2)` will succeed and return '0' if a zero-length `write(2)` is issued, but no zero-length message will be generated or sent. This option is the default for regular *Stream*, but is *not* set by default for *STREAMS*-based pipes.

SNDPIPE Issues a `{SIGPIPE}` signal to caller of `write(2)` if the caller attempts to write to a *Stream* that has received a hangup (`M_HANGUP`) or an error (`M_ERROR`). When not set, `{SIGPIPE}` will not be signalled. This option is the default for *STREAMS*-based pipes but is *not* set by default for regular *Streams*.

SNDHOLD Requests that the *Stream head* hold messages temporarily in an attempt to coalesce smaller messages into larger ones for efficiency. This feature is largely deprecated, but is supported by *Linux Fast-STREAMS*. When not set (as is

¹⁵ This setting is used with the `timod(4)` module requiring the use of the `tirdwr(4)` module for use with the `xti(3)` library.

¹⁶ This may be useful for specialized libraries or at the user's option with `timod(4)` or `sockmod(4)` modules.

¹⁷ This setting is used with the `sockmod(4)` module, or at the user's option with other modules or drivers.

¹⁸ The `RPROTCOMPRESS` option is not defined by *SVR 4.2*, but is defined by some implementations based on *SVR 4.2*.

the default), messages are sent immediately. This option is *not* set by default for any *Stream*.

5.2.3.5 Write Offset

A write offset is provided as a option to allow for reservation of bytes at the beginning of the **M_DATA** message resulting from a call to the **write(2)** system call.

The write offset can be altered by a module or driver using the **SO_WROFF** flag and *so_wroff* member of the **stroptions(9)** data structure contained in an **M_SETOPTS** message passed upstream. It is not possible for a user level process to alter the write offset using any **streamio(7)** command.

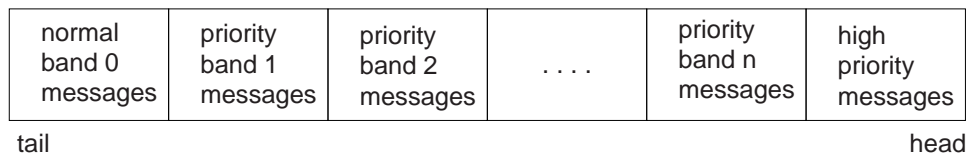
The write offset associated with a *Stream head* determines the amount of space that the *Stream head* will attempt to reserve at the beginning of the initial **M_DATA** message generated in response to the **write(2)** system call. The purpose of a write offset is to permit modules and drivers to request that bytes at the beginning of a downstream messages be reserved to permit, for example, the addition of protocol headers to the message as it passes without the need to allocate additional message blocks and prepend them.

The write offset, however, is advisory to the *Stream head* and if it cannot include the offset, a **M_DATA** message with no offset may still be generated. It is the responsibility of the module or driver to ensure that sufficient bytes are reserved at the start of a message before attempting to use them.

5.3 Queues and Priority

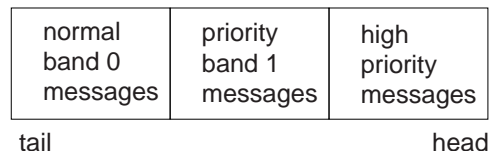
Each queue in a *Stream* has associated with it a message queue that consists of a double linked list of message blocks. Messages are normally placed onto a message queue by the queue's **put** procedure, and removed by the **service** procedure. Messages will accumulate in the message queue whenever the rate at which messages are place onto the message queue by the **put** procedure exceeds the rate at which they are removed by the **service** procedure. The **service** procedure can become blocked for a number of reasons: the *STREAMS* scheduler is delayed in invoking the **service** procedure due to higher priority system tasks; the **service** procedure is awaiting a message block necessary to complete its processing of a message; the **service** procedure is blocked by flow control forward in the *Stream*.

When a queue **service** procedure runs, it take messages off of the message queue from the head of the message queue in the order in which they appear in the queue. Messages are queued according to their priority: high priority messages appear first, followed by priority messages of descending band number, followed by normal messages in band zero. Within a band, messages are processed in the order in which they arrived at the queue (that is, on a *First-In-First-Out (FIFO)* basis). High priority messages are also processed in the order in which they arrived at the queue. This ordering within the queue is illustrated in [Figure 5.3](#).

Figure 5.3: *Message Ordering on a Queue*

When a message is placed on a queue, (e.g., by `putq(9)`), it is placed on the queue behind messages of the same priority. High priority messages are not subjected to flow control. Priority messages will affect the flow control parameters in the `qband(9)` structure associated with the band. Normal messages will affect the flow control parameter in the `queue(9)` structure. Message priority range from ‘0’ to ‘255’, where ‘0’ is the lowest queueing priority and ‘255’ the highest. High priority messages are considered to be of greater priority than all other messages.

Bands can be used for any purpose required by a service interface. For example, simple *Expedited Data* implementation can be accomplished by using one band in addition to normal messages, band ‘1’. This is illustrated in Figure 5.4.

Figure 5.4: *Message Ordering with One Priority Band*

High priority messages are considered to be of greatest priority and are not subjected to flow control. High priority messages are a rare occurrence on the typical *Stream*, and the *Stream head* only permits one high priority message (`M_PCPROTO`) to be outstanding for a user. If a high priority message arrives at the *Stream head* and one is already waiting to be read by the user, the message is discarded. High priority messages are typically handled directly from a queue’s `put` procedure, but they may also be queued to the message queue. When queue, a high priority message will always cause the `service` procedure of the queue (if any) to be scheduled for execution by the *STREAMS* scheduler. When a `service` procedure runs, and a message is retrieved from the message queue (e.g., with `getq(9)`), high priority messages will always be retrieved first. High priority messages must be acted upon immediately by a `service` procedure, it is not possible to place a high priority message back on a queue with `putbq(9)`.

5.3.1 Queue Priority Utilities

The following *STREAMS* utilities are provided to module and driver writers for use in `put` and `service` procedures. These utilities assist with handling flow control within a *Stream*.

`flushq(9)`

`flushband(9)`

These utilities provide the ability to flush specific messages from a message queue. They are discussed under [Section 7.3 \[Flush Handling\]](#), page 95, and

under [Appendix C \[Utilities\]](#), page 119. These utilities are also described in the corresponding manual page.

`canput(9)`
`bcanput(9)`
`canputnext(9)`
`bcanputnext(9)`

These utilities provide the ability to test the current or next queue for a flow control condition for normal (band zero) messages or priority messages within a message band. They are discussed under [Section 5.3.5.1 \[Flow Control\]](#), page 91, and under [Appendix C \[Utilities\]](#), page 119. These utilities are also described in the corresponding manual page.

`strqset(9)`
`strqget(9)`

These utilities provide the ability to examine and modify flow control parameters associated with a queue (`queue(9)`) or queue band (`qband(9)`). They are discussed below, and under [Appendix C \[Utilities\]](#), page 119. These utilities are also described in the corresponding manual page.

The `strqget(9)` and `strqset(9)` *STREAMS* utilities are provided to access and modify members of the `queue(9)` and `qband(9)` data structures. In general, the specific members of these data structures should not be access directly by the module writer. This restriction is necessary for several reasons:

- The size and format of the `queue(9)` and `qband(9)` structures might change, breaking binary modules compiled against the older definitions. `strqget(9)` and `strqset(9)` provide structure independent access to these members.
- On *Symetric Multi-Processing (SMP)* architectures, it may be necessary to protect access to a member of these structures to guarnatee atomicity of operations. `strqget(9)` and `strqset(9)` provide necessary locking on *SMP* architectures.

5.3.1.1 strqget(9)

A declaration for the `strqget(9)` utility is exposed by including the ‘`sys/stream.h`’ kernel header file. The prototype is as follows:

```
int strqget(queue_t *q, qfields_t what, unsigned char band, long *val);
```

Where the arguments are interpreted as follows:

<i>q</i>	Specifies the <code>queue(9)</code> structure (and indirectly the <code>qband(9)</code> structure) from which to retrieve a member.
<i>what</i>	Specifies which member to retrieve. Specific values for various members are described below.
<i>band</i>	When zero, specifies that the member is to be retrieved from the <code>queue(9)</code> structure specified by <i>q</i> ; when non-zero, the band number of the <code>qband(9)</code> structure from which to retrieve the member.

val Points to a long value into which the result is to be placed. All results are converted to a long before being written to this location.

The `qfields_t(9)` enumeration is defined as follows:

```
typedef enum qfields {
    QHIWAT,      /* hi water mark */
    QLOWAT,      /* lo water mark */
    QMAXPSZ,     /* max packet size */
    QMINPSZ,     /* min packet size */
    QCOUNT,     /* count */
    QFIRST,      /* first message in queue */
    QLAST,       /* last message in queue */
    QFLAG,       /* state */
    QBAD,        /* last (AIX and SUPER-UX) */
} qfields_t;
```

Each value of the `qfields_t` enumeration specifies a different member to be set by `strqset(9)` or retrieved by `strqget(9)`. When *band* is zero, the member to be set or retrieved is the corresponding member of the `queue(9)` structure indicated by *q*. When *band* is non-zero, the member to be set or retrieved is the corresponding member of the `qband(9)` structure, associated with *q*, of band number *band*.

QHIWAT	Set or return the high water mark (<i>q-hiwat</i> or <i>qb-hiwat</i>).
QLOWAT	Set or return the low water mark (<i>q-lowat</i> or <i>qb-lowat</i>).
QMAXPSZ	Set or return the maximum packet size (<i>q-maxpsz</i> or <i>qb-maxpsz</i>).
QMINPSZ	Set or return the minimum packet size (<i>q-minpsz</i> or <i>qb-minpsz</i>).
QCOUNT	Return the count of bytes queued (<i>q-count</i> or <i>qb-count</i>). This field is only valid for <code>strqget(9)</code> .
QFIRST	Return a pointer to the first message queued (<i>q-first</i> or <i>qb-first</i>). This field is only valid for <code>strqget(9)</code> .
QLAST	Return a pointer to the last message queued (<i>q-last</i> or <i>qb-last</i>). This field is only valid for <code>strqget(9)</code> .
QFLAG	Return the flags word (<i>q-flag</i> or <i>qb-flag</i>). This field is only valid for <code>strqget(9)</code> .

Additional information is given under [Appendix C \[Utilities\]](#), page 119, and provided in the `strqget(9)` manual page.

5.3.1.2 strqset(9)

A declaration for the `strqset(9)` utility is exposed by including the `'sys/stream.h'` kernel header file. The prototype is as follows:

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

Where the arguments are interpreted as follows:

q Specifies the `queue(9)` structure (and indirectly the `qband(9)` structure) to which to write a member.

- what* Specifies which member to write. Specific values for various members are described above under [Section 5.3.1.1 \[strqget\(9\)\]](#), page 82.
- band* When zero, specifies that the member is to be written to the `queue(9)` structure specified by *q*; when non-zero, the band number of the `qband(9)` structure to which to write the member.
- val* Specifies the `long` value to write to the member. All values are converted to a `long` to be passed in this argument.

Additional information is given under [Appendix C \[Utilities\]](#), page 119, and provided in the `strqset(9)` manual page.

5.3.2 Queue Priority Commands

Aside from the `putpmsg(2)` and `getpmsg(2)` system calls, a number of `streamio(7)` commands associated with queueing and priorities can be issued by a user level process using the `ioctl(2)` system call. The input output controls that accept a queue band or indicate a queue band event are as follows:

`I_FLUSHBAND(7)`

Flushes the *Stream* for a specified band. This `ioctl(2)` command is equivalent to the `flushq(9)` and `flushband(9)` utilities available to modules and drivers. It is discussed under [Section 7.3 \[Flush Handling\]](#), page 95.

`I_CKBAND(7)`

Checks whether a message is available to be read from a specified queue band. It is discussed below.

`I_GETBAND(7)`

Gets the priority band associated with the next message on the *Stream head* read queue. It is discussed below.

`I_CANPUT(7)`

Checks whether messages can be written to a specified queue band. This `ioctl(2)` command is equivalent to the `canput(9)` and `bcanput(9)` utilities available to modules and drivers. It is discussed under [Section 5.3.5.1 \[Flow Control\]](#), page 91.

`I_ATMARK(7)`

This `ioctl(2)` command supports *Transmission Control Protocol (TCP)* urgent data in a byte-stream. It indicates when a marked message has arrived at the *Stream head*. It is discussed below.

`I_GETSIG(7)`

`I_SETSIG(7)`

Sets the mask of events for which the *Stream head* will send a calling process a `{SIGPOLL}` or `{SIGURG}` signal. Events include `S_RDBAND`, `S_WRBAND` and `S_BANDURG`. This `ioctl(2)` command is discussed under [Section 6.1 \[Input and Output Polling\]](#), page 93.

The `streamio(7)` input output controls in the following sections are all of the form:

```
int ioctl(int fildes, int cmd, long arg);
```

5.3.2.1 I_FLUSHBAND

Flushes the *Stream* for a specified band. This `ioctl(2)` command is equivalent to the `flushq(9)` and `flushband(9)` utilities available to modules and drivers. It is discussed under [Section 7.3 \[Flush Handling\]](#), page 95.

`fildes` the *Stream* for which the command is issued;
`cmd` is 'I_FLUSHBAND'; and,
`arg` is a pointer to a `bandinfo(9)` structure.

The `bandinfo(9)` structure is exposed by including the 'sys/stropts.h' system header file. Its format and members are as follows:

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

where,

`bi_pri` the priority band to flush;
`bi_flag` how to flush: one of FLUSHR, FLUSHW or FLUSHRW.

5.3.2.2 I_CKBAND

Checks whether a message is available to be read from a specified queue band.

`fildes` the *Stream* for which the command is issued;
`cmd` is 'I_CKBAND'.
`arg` contains the band number for which to check for an available message.

5.3.2.3 I_GETBAND

Gets the priority band associated with the next message on the *Stream head* read queue.

`fildes` the *Stream* for which the command is issued;
`cmd` is 'I_GETBAND'.
`arg` is a pointer to an `int` into which to receive the band number.

5.3.2.4 I_CANPUT

The `I_CANPUT(7)` `ioctl(2)` command has the following form:

```
int ioctl(int fildes, int cmd, long arg);
```

where,

`fildes` the *Stream* for which the command is issued;

`cmd` is `I_CANPUT`.

`arg` contains the band number for which to check for flow control.

Checks whether message can be written to the queue band specified by `arg`. `arg` is an integer which contains the queue band to test for flow control. `arg` can also have the following value:

ANYBAND When this value is specified, instead of testing a specified band, `I_CANPUT(7)` tests whether any (existing) band is writable.

Upon success, the `I_CANPUT(7) ioctl(2)` command returns zero ('0') or a positive integer. The `I_CANPUT(7)` command returns false ('0') if the band cannot be written to (due to flow control), and returns true ('1') if the band is writable. Upon failure, the `ioctl(2)` call returns '-1' and sets `errno(3)` to an appropriate error number.

When the `I_CANPUT(7) ioctl(2)` command fails, it returns '-1' and sets `errno(3)` to one of the following errors:

[EINVAL] `arg` is outside the range '0' to '255' and does not represent a valid priority band, or is not **ANYBAND**.

[EIO] `fildev` refers to a *Stream* that is closing.

[ENXIO] `fildev` refers to a *Stream* that has received a hangup.

[EPIPE] `fildev` refers to a *STREAMS*-based pipe and the other end of the pipe is closed.

[ESTRPIPE] `fildev` refers to a *STREAMS*-based pipe and a write operation was attempted with no readers at the other end, or a read operation was attempted, the pipe is empty, and there are no readers/writers the other end.

[EINVAL] `fildev` refers to a *Stream* that is linked under a multiplexing driver. If a *Stream* is linked under a multiplexing driver, all `ioctl(2)` commands other than `I_UNLINK(7)` or `I_PUNLINK(7)` will return [EINVAL].

Any error received in an `M_ERROR` message indicating a persistent write error for the *Stream* will cause `I_CANPUT(7)` to fail, and the write error will be returned in `errno(3)`.

Any error number returned in `errno(3)` in response to a general `ioctl(2)` failure can also be returned in response to `I_ATMARK(7)`. See also `ioctl(2p)`.

Linux Fast-STREAMS implements the special flag, **ANYBAND**, that can be used for an `arg` value instead of the band number to check whether any existing band is writable. This is similar to the `POLLWRBAND` flag to `poll(2)`. **ANYBAND** uses the otherwise invalid band number '-1'. Portable *STREAMS* applications programs will not use the **ANYBAND** flag and will not rely upon `I_CANPUT(7)` to generate an error if passed '-1' as an invalid argument.

5.3.2.5 I_ATMARK

The `I_ATMARK(7) ioctl(2)` command has the following form:

```
int ioctl(int fildev, int cmd, long arg);
```

where,

fildev the *Stream* for which the command is issued;
cmd is 'I_ATMARK'.
arg specifies a criteria for checking for a mark.

The I_ATMARK(7) command informs the user if the current message on the *Stream head* read queue is marked by a downstream module or driver. The *arg* argument determines how the checking is done when there are multiple marked messages on the *Stream head* read queue. The possible values of the *arg* argument are as follows:

ANYMARK Determine if the message at the head of the *Stream head* read queue is marked by a downstream module or driver.
LASTMARK Determine if the message at the head of the *Stream head* read queue is the last message that is marked on the queue by a downstream module or driver.

The bitwise inclusive *OR* of the flags ANYMARK and LASTMARK is permitted.

STREAMS message blocks that have the MSGMARK flag set in the *b_flag* member of the msgb(9) structure are marked messages. Solaris also provides the MSGMARKNET and MSGNOTMARKNET flags. The use of these flags is not very clear, but Linux Fast-STREAMS could use them in the read(2) logic to determine whether the next message is marked without removing the message from the queue.

When read(2) encounters a marked message and data has already been read, the read terminates with the amount of data read. The resulting short read is an indication to the user that a marked message could exist on the read queue. (Short reads can also result from zero-byte data, or from a delimited message: one with the MSGDELIM flag set in *b_flag*). When a short read occurs, the user should test for a marked message using the ANYMARK flag to the I_ATMARK(7) ioctl(2) command. A subsequent read(2) will consume the marked message following the marked message. This can be checked by using the LASTMARK flag to the I_ATMARK(7) ioctl(2) command.

The *b_flag* member of the msgb(9) structure can have the flag, MSGMARK, set that allows a module or driver to mark a message sent to the *Stream head*. This is used to support tcp(4)'s ability to indicate 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 the *Stream head* read queue is marked, and whether it is the last marked message on the queue, with the I_ATMARK(7) ioctl(2) command. If a user is reading data from the *Stream head* and there are multiple messages on the *Stream head* read queue, and one of those messages is marked, 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. ANYMARK indicates that the user merely wants to check if the message at the head of the *Stream head* read queue is marked. LASTMARK indicates that the user wants to see if the message is the only one marked on the queue.

Upon success, the I_ATMARK(7) ioctl(2) command returns zero ('0') or a positive integer. The I_ATMARK(7) operation returns a value of true ('1') if the marking criteria is met. It returns false ('0') if the marking criteria is not met. Upon failure, the I_ATMARK(7) ioctl(2) command returns '-1' and sets errno(3) to an appropriate error number.

When the I_ATMARK(7) ioctl(2) command fails, it returns '-1' and sets errno(3) to one of the following errors:

[EINVAL] *arg* was other than ANYMARK or LASTMARK, or a bitwise-OR of the two.

Any error number returned in `errno(3)` in response to a general `ioctl(2)` failure can also be returned in response to `I_ATMARK(7)`. See also `ioctl(2p)`.

5.3.2.6 I_GETSIG

Sets the mask of events for which the *Stream* head will send a calling process a {SIGPOLL} or {SIGURG} signal. Events include `S_RDBAND`, `S_WRBAND` and `S_BANDURG`. This `ioctl(2)` command is discussed under [Section 6.1 \[Input and Output Polling\]](#), page 93.

fildev the *Stream* for which the command is issued;
cmd is 'I_GETSIG'.
arg is a pointer to a `int` to contain the retrieved event flags.

Event flags can include the following band related events:

<code>S_RDBAND</code>	a message of non-zero priority band has been placed to the <i>Stream head</i> read queue.
<code>S_WRBAND</code>	a priority band that was previously flow controlled has become available for writing (i.e., is no longer flow controlled).
<code>S_BANDURG</code>	a modifier to <code>S_RDBAND</code> to generate {SIGURG} instead of {SIGPOLL} in response to the event.

5.3.2.7 I_SETSIG

Sets the mask of events for which the *Stream* head will send a calling process a {SIGPOLL} or {SIGURG} signal. Events include `S_RDBAND`, `S_WRBAND` and `S_BANDURG`. This `ioctl(2)` command is discussed under [Section 6.1 \[Input and Output Polling\]](#), page 93.

fildev the *Stream* for which the command is issued;
cmd is 'I_SETSIG'.
arg is an integer value that contains the event flags.

Event flags can include the following band related events:

<code>S_RDBAND</code>	a message of non-zero priority band has been placed to the <i>Stream head</i> read queue.
<code>S_WRBAND</code>	a priority band that was previously flow controlled has become available for writing (i.e., is no longer flow controlled).
<code>S_BANDURG</code>	a modifier to <code>S_RDBAND</code> to generate {SIGURG} instead of {SIGPOLL} in response to the event.

5.3.3 The queue Structure

The `queue(9)` structure is exposed by including 'sys/stream.h'.

```

typedef struct queue {
    struct qinit *q_qinfo;           /* info structure for the queue */
    struct msgb *q_first;            /* head of queued messages */
    struct msgb *q_last;            /* tail of queued messages */
    struct queue *q_next;            /* next queue in this stream */
    struct queue *q_link;            /* next queue for scheduling */
    void *q_ptr;                     /* private data pointer */
    size_t q_count;                  /* number of bytes in queue */
    unsigned long q_flag;            /* queue state */
    ssize_t q_minpsz;                /* min packet size accepted */
    ssize_t q_maxpsz;                /* max packet size accepted */
    size_t q_hiwat;                  /* hi water mark for flow control */
    size_t q_lowat;                  /* lo water mark for flow control */
    struct qband *q_bandp;           /* band's flow-control information */
    unsigned char q_nband;           /* number of priority bands */
    unsigned char q_blocked;         /* number of bands flow controlled */
    unsigned char qpad1[2];          /* reserved for future use */
    /* Linux fast-STREAMS specific members */
    ssize_t q_msgs;                  /* messages on queue, Solaris counts
                                     mblks, we count msgs */
    rwlock_t q_lock;                 /* lock for this queue structure */
    int (*q_ftmsg) (mblk_t *);       /* message filter ala AIX */
} queue_t;

```

The following members are defined in *SVR 4.2*:

<i>q_qinfo</i>	points to the <i>qinit(9)</i> structure associated with this queue;
<i>q_first</i>	first message on the message queue (NULL if message queue is empty);
<i>q_last</i>	last message on the message queue (NULL if message queue is empty);
<i>q_next</i>	next queue in the <i>Stream</i> ;
<i>q_link</i>	next queue in the <i>STREAMS</i> scheduler list;
<i>q_ptr</i>	pointer to module/driver private data;
<i>q_count</i>	number of bytes of messages on the queue;
<i>q_flag</i>	queue flag bits (current state of the queue);
<i>q_minpsz</i>	minimum packet size accepted;
<i>q_maxpsz</i>	maximum packet size accepted;
<i>q_hiwat</i>	high water mark (queued bytes) for flow control;
<i>q_lowat</i>	low water mark (queued bytes) for flow control;
<i>q_bandp</i>	pointer to <i>qband(9)</i> structures associated with this queue;
<i>q_nband</i>	the number of <i>qband(9)</i> structures associated with this queue;
<i>q_blocked</i>	the number of currently blocked (flow controlled) queue bands;
<i>qpad1</i>	reserved for future use;

The following members are not defined in *SVR 4.2* and are *Linux Fast-STREAMS* specific:

<i>q_msgs</i>	number of messages on the queue;
<i>q_lock</i>	queue structure lock; and,
<i>q_ftmsg</i>	message filter ala AIX.

5.3.3.1 Using queue Information

5.3.3.2 queue Flags

```

#define QENAB          (1<< 0) /* queue is enabled to run */
#define QWANTR         (1<< 1) /* flow controlled forward */
#define QWANTW         (1<< 2) /* back-enable necessary */
#define QFULL          (1<< 3) /* queue is flow controlled */
#define QREADR         (1<< 4) /* this is the read queue */
#define QUSE           (1<< 5) /* queue being allocated */
#define QNOENB         (1<< 6) /* do not enable with putq */
#define QUP            (1<< 7) /* uni-processor emulation */
#define QBACK          (1<< 8) /* the queue has been back enabled */
#define QOLD           (1<< 9) /* module supports old style open/close */
#define QHLIST         (1<<10) /* stream head is on scan list */
#define QTOENAB        (1<<11) /* to be enabled */
#define QSYNCH         (1<<12) /* flag for queue sync */
#define QSAFE          (1<<13) /* safe callbacks needed */
#define QWELDED        (1<<14) /* flags for welded queues */
#define QSVCBUSY       (1<<15) /* service procedure running */
#define QWCLOSE        (1<<16) /* q in close wait */
#define QPROCS         (1<<17) /* putp, srvp disabled */

```

The following `queue(9)` flags are defined by *SVR 4.2*:

<code>QENAB</code>	queue is enabled to run
<code>QWANTR</code>	flow controlled forward
<code>QWANTW</code>	back-enable necessary
<code>QFULL</code>	queue is flow controlled
<code>QREADR</code>	this is the read queue
<code>QUSE</code>	queue being allocated
<code>QNOENB</code>	do not enable with <code>putq</code>
<code>QBACK</code>	the queue has been back enabled
<code>QOLD</code>	module supports old style <code>open/close</code>
<code>QHLIST</code>	stream head is on scan list

The following are not defined by *SVR 4.2*, but are used by *Linux Fast-STREAMS* and other *SVR 4.2*-based implementations:

<code>QUP</code>	uni-processor emulation
<code>QTOENAB</code>	to be enabled
<code>QSYNCH</code>	flag for queue sync
<code>QSAFE</code>	safe callbacks needed
<code>QWELDED</code>	flags for welded queues
<code>QSVCBUSY</code>	service procedure running
<code>QWCLOSE</code>	q in close wait
<code>QPROCS</code>	<code>putp</code> , <code>srvp</code> disabled

5.3.4 The `qband` Structure

The `qband(9)` structure and `qband_t(9)` type are exposed when `'sys/stream.h'` is included and are formatted and contain the following members:

```

typedef struct qband {
    struct qband *qb_next;           /* next (lower) priority band */
    size_t qb_count;                 /* number of bytes queued */
    struct msgb *qb_first;           /* first queue message in this band */
    struct msgb *qb_last;           /* last queued message in this band */
    size_t qb_hiwat;                 /* hi water mark for flow control */
    size_t qb_lowat;                 /* lo water mark for flow control */
    unsigned long qb_flag;           /* flags */
    long qb_pad1;                    /* OSF: reserved */
} qband_t;

#define qb_msgs qb_pad1

```

Where the members are interpreted as follows:

<i>qb_next</i>	points to the next (lower) priority band;
<i>qb_count</i>	number of bytes queued to this band in the message queue;
<i>qb_first</i>	the first message queued in this band (NULL if band is empty);
<i>qb_last</i>	the last message queued in this band (NULL if band is empty);
<i>qb_hiwat</i>	high water mark (in bytes queued) for this band;
<i>qb_lowat</i>	low water mark (in bytes queued) for this band;
<i>qb_flag</i>	queue band flags (see below);
<i>qb_pad1</i>	reserved for future used; and,
<i>qb_msgs</i>	same as <i>qb_padq</i> : contains the number of messages queued to the band.

Including `'sys/stream.h'` also exposes the following constants for use with the *qb_flag* member of the `qband(9)` structure:

QB_FULL	when set, indicates that the band is considered full;
QB_WANTW	when set, indicates that a preceding queue wants to write to this band; and,
QB_BACK	when set, indicates that the queue needs to be back-enabled.

5.3.4.1 Using qband Information

5.3.5 Message Processing

5.3.5.1 Flow Control

5.3.6 Scheduling

5.3.6.1 Flow Control Variables

5.3.6.2 Flow Control Procedures

5.3.6.3 The *STREAMS* Scheduler

5.4 Service Interfaces

5.4.1 Service Interface Benefits

5.4.2 Service Interface Library Example

5.4.2.1 Accessing the Service Provider

5.4.2.2 Closing the Service Provider

5.4.2.3 Sending Data to the Service Provider

5.4.2.4 Receiving Data

5.4.2.5 Module Service Interface Example

5.5 Message Allocation

5.5.1 Recovering From No Buffers

5.6 Extended Buffers

6 Polling

6.1 Input and Output Polling

6.2 Controlling Terminal

7 Modules and Drivers

7.1 Environment

7.2 Input-Output Control

7.3 Flush Handling

7.4 Driver-Kernel Interface

7.5 Design Guidelines

8 Modules

8.1 Module

8.2 Module Flow Control

8.3 Module Design Guidelines

9 Drivers

9.1 External Device Numbers

9.2 Internal Device Numbers

9.3 spec File System

9.4 Clone Device

9.5 Named STREAMS Device

9.6 Driver

9.7 Cloning

9.8 Loop-Around Driver

9.9 Driver Design Guidelines

10 Multiplexing

10.1 Multiplexors

10.2 Connecting and Disconnecting Lower Stream

10.3 Multiplexor Construction Example

10.4 Multiplexing Driver

10.5 Persistent Links

10.6 Multiplexing Driver Design Guidelines

11 Pipes and FIFOs

11.1 Pipes and FIFOs

11.2 Flushing Pipes and FIFOs

11.3 Named Streams

11.4 Unique Connections

12 Terminal Subsystem

12.1 Terminal Subsystem

12.2 Pseudo-Terminal Subsystem

13 Synchronization

13.1 MT Configuration

13.2 Asynchronous Entry Points

13.3 Asynchronous Callbacks

13.4 Synchronous Entry Points

13.5 Synchronous Callbacks

13.6 STREAMS Framework Integrity

13.7 MP Message Ordering

13.8 MP-UNSAFE Modules

13.9 MP Put and Service Procedures

13.10 MP Timeout and Buffer Callbacks

13.11 MP Open and Close Procedures

13.12 MP Module Unloading

13.13 MP Locking

13.14 MP Asynchronous Callbacks

13.15 Stream Integrity

14 Reference

14.1 Files

14.2 System Modules

14.3 System Drivers

14.4 System Calls

14.5 Input-Output Controls

14.6 Module Entry Points

14.7 Structures

14.8 Registration

14.9 Message Handling

14.10 Queue Handling

14.11 Miscellaneous Functions

14.12 Extensions

14.13 Compatibility

15 Conformance

15.1 SVR 4.2 Compatibility

15.2 AIX Compatibility

15.3 HP-UX Compatibility

15.4 OSF/1 Compatibility

15.5 UnixWare Compatibility

15.6 Solaris Compatibility

15.7 SUX Compatibility

15.8 UXP Compatibility

15.9 LiS Compatibility

16 Portability

16.1 Core Function Support

16.2 SVR 4.2 Portability

16.3 AIX Portability

16.4 HP-UX Portability

16.5 OSF/1 Portability

16.6 UnixWare Portability

16.7 Solaris Portability

16.8 SUX Portability

16.9 UXP Portability

16.10 LiS Portability

Appendix A Data Structures

A.1 Stream Structures

A.2 Queue Structures

A.3 Message Structures

A.4 Input Output Control Structures

A.5 Link Structures

A.6 Options Structures

Appendix B Message Types

B.1 Message Type

B.2 Ordinary Messages

B.3 High Priority Messages

Appendix C Utilities

Appendix D Debugging

Appendix E Configuration

Appendix F Administration

F.1 Administrative Utilities

F.2 System Controls

F.3 /proc File System

Appendix G Examples

G.1 Module Example

G.2 Driver Example

Appendix H Copying

H.1 GNU General Public License

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Copyright © 1989, 1991 Free Software Foundation, Inc.
675 Mass Ave, Cambridge, MA 02139, USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

H.1.1 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in

effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

H.1.2 Terms and Conditions for Copying, Distribution and Modification

1. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

3. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:
 - a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
 - b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
 - c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

4. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
 - a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
 - b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
 - c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

5. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or

distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

6. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
8. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

9. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries

not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

10. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and “any later version”, you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

11. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

12. **BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.**
13. **IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.**

END OF TERMS AND CONDITIONS

H.1.3 How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

```
one line to give the program's name and an idea of what it does.
Copyright (C) 19yy name of author
```

```
This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.
```

```
This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.
```

```
You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type 'show w'. This is free software, and you are welcome
to redistribute it under certain conditions; type 'show c'
for details.
```

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than ‘show w’ and ‘show c’; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright
interest in the program 'Gnomovision'
(which makes passes at compilers) written
by James Hacker.
```

```
signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

H.2 GNU Free Documentation License

GNU FREE DOCUMENTATION LICENSE

Version 1.1, March 2000

Copyright © 2000 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

H.2.1 Preamble

The purpose of this License is to make a manual, textbook, or other written document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

H.2.2 Terms and Conditions for Copying, Distribution and Modification

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related

matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers

that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. In any section entitled "Acknowledgments" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgments and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement

made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled “History” in the various original documents, forming one section entitled “History”; likewise combine any sections entitled “Acknowledgments”, and any sections entitled “Dedications”. You must delete all sections entitled “Endorsements.”

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an “aggregate”, and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document’s

Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

END OF TERMS AND CONDITIONS

H.2.3 How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.1
or any later version published by the Free Software Foundation;
with the Invariant Sections being list their titles, with the
Front-Cover Texts being list, and with the Back-Cover Texts being list.
A copy of the license is included in the section entitled ‘‘GNU
Free Documentation License’’.
```

If you have no Invariant Sections, write “with no Invariant Sections” instead of saying which ones are invariant. If you have no Front-Cover Texts, write “no Front-Cover Texts” instead of “Front-Cover Texts being *list*”; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Glossary

<i>anchor</i>	A <i>STREAMS</i> locking mechanism that prevents the removal of <i>STREAMS</i> modules with the <code>I_POP</code> <code>ioctl</code> . Anchors are placed on <i>STREAMS</i> modules by adding the ‘ <code>[anchor]</code> ’ flag to <code>autopush(8)</code> configuration files or directly with the <code>I_ANCHOR</code> <code>ioctl</code> .
<i>autopush</i>	A <i>STREAMS</i> mechanism that enables a pre-specified list of modules to be pushed automatically onto a <i>Stream</i> when a <i>STREAMS</i> device is opened. This mechanism is used only for administrative purposes.
<i>back-enable</i>	To enable (by <i>STREAMS</i>) a preceding blocked queue’s <code>service</code> procedure when <i>STREAMS</i> determines that a succeeding queue has reached its low-water mark.
<i>blocked</i>	A queue’s <code>service</code> procedure that cannot be enabled due to flow control.
<i>clone device</i>	A <i>STREAMS</i> device that returns an unused major/minor device number when initially opened, rather than requiring the minor device to be specified by name in the <code>open</code> call.
<i>close procedure</i>	A routine that is called when a module is popped from a <i>Stream</i> or when a driver is closed. A pointer to this procedure is specified in the <code>qi_qopen</code> member of the <code>queue(9)</code> structure associated with the read side of the module’s queue pair.
<i>control Stream</i>	A <i>Stream</i> above a multiplexing driver used to establish lower multiplexer connections. Multiplexed <i>Stream</i> configurations are maintained through the controlling <i>Stream</i> to a multiplexing driver.
<i>Device Driver Interface</i>	An interface that facilitates driver portability across different <i>UNIX</i> system versions.
<i>device driver</i>	A <i>Stream</i> component whose principle functions are handling an associated physical device and transforming data and information between the external interface and the <i>Stream</i> .

<i>Driver Kernel Interface</i>	An interface between the <i>UNIX</i> 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.
<i>downstream</i>	A direction of data flow going from the <i>Stream head</i> toward a driver. Also called the <i>write-side</i> and <i>output-side</i> .
<i>driver</i>	A module that forms the <i>Stream end</i> . It can be a device driver or a pseudo-device driver. It is a required component in <i>STREAMS</i> (except in <i>STREAMS</i> -based pipes and FIFOs), and is physically identical to a module. It typically handles data transfer between the kernel and a device and does little or no processing of data.
<i>enable</i>	A term used to describe scheduling of a queue's service procedure.
<i>FIFO</i>	<i>First In, First Out</i> . A term used in <i>STREAMS</i> for named pipes. This term is also used in queue scheduling.
<i>flow control</i>	A <i>STREAMS</i> mechanism that regulates the rate of message transfer within a <i>Stream</i> and from user space into a <i>Stream</i> .
<i>hardware emulation module</i>	A module required when the terminal line discipline is on a <i>Stream</i> but there is no terminal driver at the <i>Stream end</i> . This module recognizes all termio(7) ioctl s necessary to support terminal semantics specified by termio(9) and termios(9) .
<i>input side</i>	A direction of data flow going from a driver toward the <i>Stream head</i> . Also called <i>read-side</i> and <i>upstream</i> .
<i>line discipline</i>	A <i>STREAMS</i> module that performs termio(7) canonical and non-canonical processing. It shares some termio(7) processing with a driver in a <i>STREAMS</i> terminal subsystem.
<i>lower Stream</i>	A <i>Stream</i> connected beneath a multiplexing pseudo-device driver, by means of an I_LINK or I_PLINK ioctl . The far end of a lower <i>Stream</i> terminates at a device driver or another multiplexer driver.
<i>master driver</i>	A <i>STREAMS</i> -based device supported by the pseudo-terminal subsystem. It is the controlling part of the pseudo-terminal subsystem (also called ' ptm ').

<i>message</i>	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.
<i>message block</i>	A triplet consisting of a data buffer and associated control structures, a <code>msgb(9)</code> structure, a <code>datab(9)</code> structure. It carries data or information, as identified by its message type, in a <i>Stream</i> .
<i>message queue</i>	A linked list of zero or more messages connected together.
<i>message type</i>	A enumerated set of values identifying the contents of a message.
<i>module</i>	A defined set of kernel-level routines and data structure used to process data, status, and control information on a <i>Stream</i> . It is an optional element, but there can be many modules in one <i>Stream</i> . It consists of a pair of queues (read queue and write queue), and it communicates to other components in a <i>Stream</i> by passing messages.
<i>multiplexer</i>	A <i>STREAMS</i> mechanism that allows message to be routed among multiple <i>Streams</i> in the kernel. A multiplexing configuration includes at least one multiplexing pseudo-device driver connected to one or more upper <i>Streams</i> and one or more lower <i>Streams</i> .
<i>named Stream</i>	A <i>Stream</i> , typically a pipe, with a name associated with it by way of a call to <code>fattach(3)</code> (that is, a <code>mount(2)</code> operation). This is different from a named pipe (FIFO) in two ways: a named pipe (FIFO) is unidirectional while a named <i>Stream</i> is bidirectional; a name <i>Stream</i> need not refer to a pipe, but can be another type of <i>Stream</i> .
<i>open routine</i>	A procedure in each <i>STREAMS</i> driver and module called by <i>STREAMS</i> on each <code>open</code> system call made on the <i>Stream</i> . A module's <code>open</code> procedure is also called when the module is pushed.
<i>packet mode</i>	A feature supported by the <i>STREAMS</i> -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.

<i>persistent link</i>	A connection below a multiplexer that can exist without having an open controlling <i>Stream</i> associated with it.
<i>pipe</i>	See <i>STREAMS</i> -based pipe.
<i>pop</i>	A term used when a module that is immediately below the <i>Stream</i> head is removed.
<i>pseudo-device driver</i>	A software driver, not directly associated with a physical device, that performs functions internal to a <i>Stream</i> such as a multiplexer or <code>log(4)</code> driver.
<i>pseudo-terminal subsystem</i>	A user interface identical to a terminal subsystem except that there is a process in place of a hardware device. It consists of at least a master device, slave device, line discipline module, and hardware emulation module.
<i>push</i>	A term used when a module is inserted in a <i>Stream</i> immediately below the <i>Stream head</i> .
<i>pushable module</i>	A module put between the <i>Stream head</i> and driver. It performs intermediate transformations on messages flowing between the <i>Stream head</i> and driver. A driver is a non-pushable module.
<i>put procedure</i>	A routine in a module or driver associated with a queue that receives messages from the preceding queue. It is the single entry point into a queue from a preceding queue. It may perform processing on the message and will then generally either queue the message for subsequent processing by this queue's <code>service</code> procedure, or will pass the message to the <code>put</code> procedure of the following queue (using <code>putnext(9)</code>).
<i>queue</i>	A data structure that contains status information, a pointer to routines processing message, and pointers for administering a <i>Stream</i> . It typically contains pointer to <code>put</code> and <code>service</code> procedures, a message queue, and private data.
<i>read-side</i>	A direction of data flow going from a driver toward the <i>Stream head</i> . Also called <i>upstream</i> and <i>input-side</i> .
<i>read queue</i>	A message queue in a module or driver containing messages moving <i>upstream</i> . Associated with the <code>read(2)</code> system call and input from a driver.

<i>remote mode</i>	A feature available with the pseudo-terminal subsystem. It is used for applications that perform the canonical and echoing functions normally done by line discipline module and TTY driver. It enables applications on the master side to turn off the canonical processing.
<i>STREAMS Administrative Driver</i>	A <i>STREAMS</i> Administrative Driver that provides an interface to the <code>autopush(8)</code> mechanism.
<i>schedule</i>	To place a queue on the internal list of queues that will subsequently have their service procedure called by the <i>STREAMS</i> scheduler. <i>STREAMS</i> scheduling is independent of <i>Linux</i> process scheduling.
<i>service interface</i>	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 <i>Stream</i> /user boundary, the primitives are typically contained in the control part of a message; within a <i>Stream</i> , in <code>M_PROTO</code> or <code>M_PCPROTO</code> message blocks.
<i>service procedure</i>	A module or driver routine associated with a queue that receives messages queue for it by the <code>put</code> procedure is called by the <i>STREAMS</i> scheduler. It may perform processing on the message and generally passes the message to the <code>put</code> procedure of the following queue.
<i>service provider</i>	An entity in a service interface that responds to request primitives from the service user with response and event primitives.
<i>service user</i>	An entity in a service interface that generates request primitives for the service provider and consumes response and event primitives.
<i>slave driver</i>	A <i>STREAMS</i> -based device supported by the pseudo-terminal subsystem. It is also called ‘ <code>pts</code> ’ and works with a line discipline module and hardware emulation module to provide an interface to a user process.
<i>standard pipe</i>	A mechanism for the unidirectional flow of data between two processes where data written by one process becomes data read by the other process.

<i>Stream</i>	A kernel level aggregate created by connecting <i>STREAMS</i> components, resulting from an application of the <i>STREAMS</i> mechanism. The primary components are the <i>Stream head</i> , the driver (or <i>Stream end</i>), and zero or more pushable modules between the <i>Stream head</i> and driver.
<i>STREAMS-based pipe</i>	A mechanism used for bidirectional data transfer implemented using <i>STREAMS</i> , and sharing the properties of <i>STREAMS</i> -based devices.
<i>Stream end</i>	A <i>Stream</i> component furthest from the user process that contains a driver.
<i>Stream head</i>	A <i>Stream</i> component closest to the user process. It provides the interface between the <i>Stream</i> and the user process.
<i>STREAMS</i>	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 <i>STREAMS</i> mechanism includes integral functions, utility routines, kernel facilities, and a set of structures.
<i>TTY driver</i>	A <i>STREAMS</i> -based device used in a terminal subsystem.
<i>upper stream</i>	A <i>Stream</i> that terminates above a multiplexing driver. The beginning of an upper <i>Stream</i> originates at the <i>Stream head</i> or another multiplexing driver.
<i>upstream</i>	A direction of data flow going from a driver toward the <i>Stream head</i> . Also called <i>read-side</i> and <i>input side</i> .
<i>water mark</i>	A limit value used in flow control. Each queue has a high-water mark and a low-water mark. The high-water mark value indicates the upper limit related to the number of bytes contained on the queue. When the queued character reaches its high water mark, <i>STREAMS</i> causes another queue that attempts to send a message to this queue to become blocked. When the characters in this queue are reduced to the low-water mark value, the other queue is unblocked by <i>STREAMS</i> .

<i>write queue</i>	A message queue in a module or driver containing messages moving downstream. Associated with the <code>write(2)</code> system call and output from a user process.
<i>write-side</i>	A direction of data flow going from the <i>Stream head</i> toward a driver. Also called downstream and output side.

List of Figures

Figure 1.1: <i>Simple Stream</i>	14
Figure 1.2: <i>STREAMS-based Pipe</i>	15
Figure 1.3: <i>STREAMS-based FIFO (named pipe)</i>	15
Figure 1.4: <i>Stream to Communications Driver</i>	20
Figure 1.5: <i>A Message</i>	23
Figure 1.6: <i>Messages on a Message Queue</i>	24
Figure 1.7: <i>A Stream in More Detail</i>	26
Figure 1.8: <i>Many-to-one Multiplexor</i>	29
Figure 1.9: <i>One-to-many Multiplexor</i>	29
Figure 1.10: <i>Many-to-many Multiplexor</i>	30
Figure 1.11: <i>Internet Multiplexing Stream</i>	31
Figure 1.12: <i>Multiplexing Stream</i>	32
Figure 1.13: <i>Protocol Module Portability</i>	34
Figure 1.14: <i>Protocol Substitution</i>	35
Figure 1.15: <i>Protocol Migration</i>	36
Figure 1.16: <i>Module Reusability</i>	37
Figure 3.1: <i>Upstream and Downstream Stream Construction</i>	43
Figure 3.2: <i>Stream Queue Relationship</i>	44
Figure 3.3: <i>Opened STREAMS-based Driver</i>	46
Figure 3.4: <i>Opened STREAMS-based FIFO</i>	48
Figure 3.5: <i>Created STREAMS-based Pipe</i>	49
Figure 3.6: <i>Case Converter Module</i>	54
Figure 4.1: <i>Put Procedure Example</i>	60
Figure 5.1: <i>Message Form and Linkage</i>	71
Figure 5.2: <i>Data Buffer References</i>	72
Figure 5.3: <i>Message Ordering on a Queue</i>	81
Figure 5.4: <i>Message Ordering with One Priority Band</i>	81

List of Figures

List of Listings

Listing 1.1: <i>Basic Operations</i>	19
Listing 3.1: <i>Inserting Modules Example</i>	53
Listing 3.2: <i>Inserting Modules Example (cont'd)</i>	53
Listing 3.3: <i>Inserting Modules Example (cont'd)</i>	53
Listing 3.4: <i>Module and Driver Control Example</i>	55
Listing 3.5: <i>Module and Driver Control Example (cont'd)</i>	57

Index

A

allocb(9) 68
 anchor 143
 ANYBAND 86
 ANYMARK 87, 88
 autopush 143
 autopush(8) 143, 147

B

b_band 68, 71, 74, 76, 77
 b_cont 68, 71, 72
 b_datap 68
 b_flag 68, 69, 87
 b_next 68, 71
 b_pad1 68
 b_pad2 68
 b_prev 68, 71
 b_rptr 68, 72
 b_wptr 68, 72
 back-enable 143
 bandinfo(9) 85
 bcanput(9) 82, 84
 bcanputnext(9) 82
 blocked 143
 buf 75, 76

C

canput(9) 82, 84
 canputnext(9) 82
 cdevsw(9) 47
 clone device 143
 close procedure 143
 close(2) 6, 12, 18, 20, 28, 42, 51, 73
 contributors 1
 control Stream 143
 copyb(9) 23, 72
 copymsg(9) 23, 72
 credits 1

D

D_CLONE 47
 datab(9) 22
 datab(9) 23
 datab(9) 65, 67, 68, 145
 db_base 70, 72
 db_cache 70
 db_class 70
 db_filler 71

db_filler2 70
 db_freep 70
 db_frtnp 70
 db_iswhat 70
 db_lim 70, 73
 db_msgaddr 71
 db_pad 70
 db_ref 70, 71, 72
 db_size 70
 db_type 70, 71
 db_users 71
 DELETE 56
 device driver 143
 Device Driver Interface 143
 document abstract 3
 document audience 3
 document disclaimer 5
 document information 3
 document intent 3
 document notice 3
 document objective 3
 document revisions 4
 downstream 144
 driver 144
 Driver Kernel Interface 144
 dupb(9) 23, 72
 dupmsg(9) 23, 72

E

EBADMSG 79
 EINVAL 86, 88
 EIO 86
 enable 144
 ENXIO 54, 86
 EPIPE 86
 errno(3) 86, 87, 88
 esballoc(9) 68
 ESTRPIPE 86
 ETIME 56
 exit(2) 18, 20, 57

F

f_inode 47
 fattach(3) 21, 51, 145
 fattach(8) 51
 fdetach(3) 22, 51
 FIFO 144
 file 47, 48, 50, 52
 flow control 144
 flushband(9) 81, 84, 85

Index

flushq(9) 81, 84, 85
FLUSHR 85
FLUSHRW 85
FLUSHW 85
FMNAMESZ 54
freemsg(9) 23

G

getmsg(2) 12, 17, 18, 22, 41, 42, 65, 73, 74, 75,
76, 77
getmsg(2p) 76
getpmsg(2) 84
getpmsg(2p) 77
getpmsg(2s) 12, 17, 18, 22, 41, 42, 66, 73, 74, 76,
77
getq(9) 81

H

hardware emulation module 144

I

I_ANCHOR 143
I_ATMARK 84, 86, 87, 88
I_CANPUT 84, 85, 86
I_CKBAND 84
I_FLUSHBAND 84
I_GETBAND 84
I_GETSIG 84
I_GRDOPT 78
I_GWROPT 78, 79
I_LINK 21, 30, 51, 144
i_pipe 47
I_PLINK 30, 144
I_POP 28, 50, 51, 57, 143
I_PUNLINK 30, 86
I_PUSH 28, 50, 53, 54, 57
I_SETSIG 84
I_SRDOPT 78
I_STR 54, 55, 56
I_SWROPT 78, 79
I_UNLINK 22, 30, 51, 86
ic_cmd 55, 56
ic_dp 56
ic_len 56
ic_timeout 55, 56
inode 47, 48, 49, 50, 52
input side 144
int 85, 88
ioc_cmd 56
ioctl(2) 53
ioctl 57

ioctl(2) 6, 12, 16, 22, 28, 41, 42, 50, 53, 54, 55,
56, 57, 66, 67, 78, 79, 84, 85, 86, 87, 88
ioctl(2p) 41, 86, 88

K

kmem_alloc(9) 68

L

LASTMARK 87, 88
len 75, 76
license, FDL 135
license, GNU Free Documentation License 135
license, GNU General Public License 129
license, GPL 129
licensing 3
line discipline 144
log(4) 146
long 83, 84
lower Stream 144

M

M_BACKDONE 67
M_BACKWASH 66
M_BREAK 66
M_COPYIN 67
M_COPYOUT 67
M_CTL 66, 67
M_DATA 17, 22, 56, 65, 66, 67, 71, 73, 74, 75, 79,
80
M_DELAY 66
M_DONTPLAY 67
M_ERROR 67, 79, 86
M_EVENT 66, 67
M_FLUSH 67
M_HANGUP 67, 79
M_HPDATA 67, 73, 75
M_IOCACK 67
M_IOCDATA 67
M_IOCNAK 67
M_IOCTL 51, 56, 66, 71
M_LETSPLAY 67
M_NOTIFY 67
M_PASSFP 66
M_PCCTL 67
M_PCEVENT 67
M_PCPROTO 22, 25, 65, 67, 73, 74, 75, 79, 81, 147
M_PCRSE 67
M_PCSETOPTS 67
M_PCSIG 67
M_PCTTY 67
M_PROTO 17, 22, 25, 65, 66, 73, 74, 75, 79, 147
M_READ 67, 78

M_RSE 66
 M_SETOPTS 66, 67, 77, 78, 79, 80
 M_SIG 66
 M_START 67
 M_STARTI 67
 M_STOP 67
 M_STOPI 67
 M_TRAIL 66
 M_UNHANGUP 67
 master driver 144
 maxlen 74, 76
 message 145
 message block 145
 message queue 145
 message type 145
 mknod(2) 17
 module 145
 module_info 44
 module_info(9) 43, 44
 module_init 43, 47, 50
 module_init(9) 44
 module_stat 43
 module_stat 44
 module_stat(9) 43, 44
 mount(2) 145
 MSG_ANY 77
 MSG_BAND 77
 MSG_HIPRI 77
 MSGATTEN 69
 msgb(9) 22
 msgb(9) 23
 msgb(9) 65, 67, 68, 71, 87, 145
 MSGCOMPRESS 69
 MSGDELIM 69, 87
 MSGLOG 69
 MSGMARK 69, 87
 MSGMARKNET 87
 MSGMARKNEXT 69
 MSGNOGET 69
 MSGNOLOOP 69
 MSGNOTIFY 69
 MSGNOTMARKNET 87
 MSGNOTMARKNEXT 69
 multiplexer 145

N

named Stream 145
 NULL 46, 47, 61, 72, 89, 91

O

O_NDELAY 20, 52
 O_NDLEAY 20
 O_NONBLOCK 20, 52

open routine 145
 open(2) ... 6, 12, 18, 19, 28, 29, 30, 32, 41, 42, 45,
 47, 48, 49, 53, 73, 74, 75
 organization 6

P

packet mode 145
 persistent link 146
 pipe 146
 pipe(2) 18, 29, 42, 45, 49, 50, 74, 75
 poll(2) 12, 41, 42, 86
 POLLRDBAND 41
 POLLWRBAND 86
 pop 146
 private_data 47
 pseudo-device driver 146
 pseudo-terminal subsystem 146
 push 146
 pushable module 146
 put procedure 146
 put(9) 21, 22, 23, 28, 59, 61, 62, 65
 putbq(9) 21, 23, 81
 putctl(9) 59
 putctl1(9) 59
 putctl2(9) 59
 putmsg(2) ... 12, 17, 18, 22, 41, 42, 66, 73, 74, 75,
 76
 putmsg(2p) 75
 putnext(9) 21, 22, 23, 26, 59, 61, 65, 146
 putnextctl(9) 59
 putnextctl1(9) 59
 putnextctl2(9) 59
 putpmsg(2) 84
 putpmsg(2p) 76
 putpmsg(2s) .. 12, 17, 18, 22, 41, 42, 66, 73, 74, 76
 putq(9) 21, 23, 60, 61, 62, 81

Q

q_bandp 89
 q_blocked 89
 q_count 83, 89
 q_first 83, 89
 q_flag 83, 89
 q_ftmsg 89
 q_hiwat 78, 83, 89
 q_init 47, 50
 q_last 83, 89
 q_link 89
 q_lock 89
 q_lowat 83, 89
 q_maxpsz 78, 83, 89
 q_minpsz 78, 83, 89
 q_msgs 89

Index

q_nband..... 89
q_next..... 47, 49, 50, 89
q_ptr..... 89
q_qinfo..... 89
QB_BACK..... 91
qb_count..... 83, 91
qb_first..... 83, 91
qb_flag..... 83, 91
QB_FULL..... 91
qb_hiwat..... 83, 91
qb_last..... 83, 91
qb_lowat..... 83, 91
qb_maxpsz..... 83
qb_minpsz..... 83
qb_msgs..... 91
qb_next..... 91
qb_pad1..... 91
qb_padq..... 91
QB_WANTW..... 91
QBACK..... 90
qband(9)..... 25, 43
qband(9)..... 45
qband(9)..... 81, 82, 83, 84, 89, 90, 91
qband_t(9)..... 90
QCOUNT..... 83
QENAB..... 90
qfields_t..... 83
qfields_t(9)..... 83
QFIRST..... 83
QFLAG..... 83
QFULL..... 90
QHIWAT..... 83
QHLIST..... 90
qi_lowat..... 78
qi_mininfo..... 47
qi_putp..... 59, 60, 61
qi_qadmin..... 59
qi_qclose..... 51, 59, 73
qi_qopen..... 47, 51, 59, 73, 143
qi_srvp..... 59, 61
qinit..... 46, 47, 50, 51
qinit(9)..... 43
qinit(9)..... 44
qinit(9)..... 59, 60, 61, 89
QLAST..... 83
QLOWAT..... 83
QMAXPSZ..... 83
QMINPSZ..... 83
QNOENB..... 90
QOLD..... 90
qpad1..... 89
QPROCS..... 90
QREADR..... 90
qreply(9)..... 22, 23, 26, 59, 65
QSAFE..... 90
QSVCBUSY..... 90

QSYNCH..... 90
QTOENAB..... 90
queue..... 47, 50, 51
queue..... 146
queue(9)..... 24, 25, 43
queue(9)..... 44, 45
queue(9)..... 59, 60, 61, 62, 81, 82, 83, 84, 88, 90, 143
QUP..... 90
QUSE..... 90
QWANTR..... 90
QWANTW..... 90
QWCLOSE..... 90
QWELDED..... 90

R

read queue..... 146
read(2)..... 6, 12, 17, 18, 20, 22, 41, 42, 65, 73, 74, 78, 79, 87, 146
read-side..... 146
readv(2)..... 78
remote mode..... 147
RFILL..... 78
RMSGD..... 78
RMSGN..... 78
RNORM..... 78
RPROCMPRESS..... 79
RPROTCMPRESS..... 79
RPROTDAT..... 79
RPROTDIS..... 79
RPROTNORM..... 78, 79

S

S_BANDURG..... 84, 88
S_IFIFO..... 50
S_RDBAND..... 84, 88
S_WRBAND..... 84, 88
schedule..... 147
sd_file..... 47
sd_inode..... 47
service interface..... 147
service procedure..... 147
service provider..... 147
service user..... 147
sfx(4)..... 49
SIGPIPE..... 50, 79
SIGPOLL..... 84, 88
SIGURG..... 84, 88
slave driver..... 147
SNDHOLD..... 79
SNDPIPE..... 50, 79
SNDZERO..... 50, 79
snode..... 47

so_readopt 78
 SO_READOPT 78
 so_wroff 80
 SO_WROFF 80
 sockmod(4) 79
 sponsors 1
 st_muxrinit 46
 st_muxwinit 46
 st_rdinit 46
 st_wrinit 46
 standard pipe 147
 stdata 47, 48, 49, 50, 52
 stdata(9) 47
 strbuf(5) 74, 75, 76
 Stream 148
 Stream end 148
 Stream head 148
 streamio(7) 8, 22, 30, 41, 57, 66, 80, 84, 85
 STREAMS 148
 STREAMS Administrative Driver 147
 STREAMS(9) 3, 8
 STREAMS-based pipe 148
 streamtab 45, 46, 47, 50
 streamtab(9) 45
 strioctl 55
 stroptions(9) 78, 80
 strqget(9) 82, 83
 strqset(9) 82, 83, 84
 sys/stream.h 65, 68, 69, 82, 83, 88, 90, 91
 sys/stropts.h 55, 74, 75, 76, 85

T

tcp(4) 87

termio(7) 144
 termio(9) 144
 termios(9) 144
 timod(4) 79
 tirdwr(4) 79
 TTY driver 148

U

UNIX System V Release 3.0 11
 UNIX System V Release 4 11
 UNIX System V Release 4.2 11
 upper stream 148
 upstream 148

V

vnode 47
 vnodes 50

W

water mark 148
 write queue 149
 write(2) ... 6, 12, 17, 18, 20, 22, 41, 42, 65, 73, 74,
 79, 80, 149
 write-side 149

X

XCASE 56
 xti(3) 79

