

Linux Fast-STREAMS STREAMS Programmer's Guide

Version 0.7a Edition 3
Updated 2005-09-14
Package streams-0.7a.3

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 3 of the Linux Fast-STREAMS documentation, and is consistent with streams 0.7a. This manual 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 manual provided the copyright notice and this permission notice are preserved on all copies.

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

Short Contents

Acknowledgements	1
1 Introduction	3
2 Overview of Linux Fast-STREAMS	15
3 STREAMS Mechanism	37
4 STREAMS Processing Routines	49
5 STREAMS Messages	57
6 Polling and Signalling	95
7 Overview of STREAMS Modules and Drivers	105
8 STREAMS Modules	139
9 STREAMS Drivers	149
10 STREAMS Multiplexing	171
11 STREAMS-based Pipes and FIFOs	195
12 STREAMS-based Terminal Subsystem	205
13 Reference	223
14 Examples	245
15 Device Numbers	247
16 Multi-Threading	251
17 Administration	255
A STREAMS Data Structures	275
B STREAMS Message Types	281
C STREAMS Utilities	295
D STREAMS Debugging	309
E STREAMS Configuration	323
F Conformance	329
G Portability	331
H Copying	347
Index	361

Table of Contents

Acknowledgements	1
Sponsors	1
Contributors	1
1 Introduction	3
1.1 What is <i>STREAMS</i> ?	3
1.2 Does Linux have <i>STREAMS</i> ?	3
1.2.1 Linux <i>STREAMS</i> (LiS)	3
1.2.2 Linux Fast- <i>STREAMS</i> (LfS)	4
1.2.3 <i>STREAMS</i> for FreeBSD	4
1.2.4 Open <i>STREAMS</i> ?	4
1.3 Why <i>STREAMS</i> ?	4
1.3.1 <i>STREAMS</i> versus Sockets	4
1.3.2 Benefits of <i>STREAMS</i>	6
1.3.2.1 Standardized Service Interfaces	6
1.3.2.2 Manipulating Modules	6
1.3.2.3 Protocol Portability	7
1.3.2.4 Protocol Substitution	7
1.3.2.5 Protocol Migration	7
1.3.2.6 Module Reusability	7
1.3.3 Criticism of <i>STREAMS</i>	7
1.3.4 Realities of <i>STREAMS</i>	10
1.4 Why Fast?	10
1.5 Why Linux?	11
1.6 Why Compatibility?	11
1.6.1 Intel Binary Compatibility Suite (iBCS)	12
1.6.1.1 OpenGroup Specifications	12
UNIX 03 Compliance	12
UNIX 98 Compliance	13
UNIX 95 Compliance	13
1.6.2 Device Driver Interface (DDI)	13
2 Overview of Linux Fast-<i>STREAMS</i>	15
2.1 <i>STREAMS</i> Definitions	15
2.2 Concepts	16
2.2.1 Stream Administration	25
2.2.2 Driver Switch Table	25
2.2.3 Module Switch Table	26
2.2.4 Stream Table	27
2.2.5 Queue Initialization	27
2.2.6 Module Information	28
2.2.7 Module Statistics	28

2.2.8	Stream Head	29
2.2.9	Queue	30
2.2.10	Queue Band	30
2.2.11	Message Block	30
2.2.12	Data Block	31
2.2.12.1	Data Block Types	31
2.2.13	Data Buffer	31
2.2.14	User Credentials	31
2.3	Application Interface	31
2.3.1	System Calls	31
2.4	Kernel Level Facilities	32
2.4.1	Stream Head	32
2.4.2	Modules	32
2.4.3	Drivers	32
2.4.4	Messages	32
2.4.4.1	Message Types	33
2.4.5	Message Queueing Priority	34
2.4.6	Queues	34
2.4.7	Multiplexing	34
2.4.8	Multithreading	34
2.5	Subsystems	34
2.5.1	Logging	34
2.5.2	Administrative Driver	35
2.5.3	Terminal I/O	35
2.5.4	Pipes	35
2.5.5	FIFOs	35
2.5.6	Networking	36
3	STREAMS Mechanism	37
3.1	STREAMS Mechanism Overview	37
3.1.1	STREAMS System Calls	37
3.2	STREAMS Stream Construction	38
3.2.1	Opening a STREAMS Device File	40
3.2.2	Creating a STREAMS-based Pipe	43
3.2.3	Adding and Removing Modules	43
3.2.4	Closing the Stream	44
3.2.5	Stream Construction Example	45
3.2.5.1	Inserting Modules	45
3.2.5.2	Module and Driver Control	46

4	STREAMS Processing Routines	49
4.1	STREAMS Put and Service Procedures	49
4.1.1	Put Procedure	49
4.1.2	Service Procedure	50
4.2	An Asynchronous Stream Example	50
4.2.1	Read-Side Processing	53
4.2.1.1	Driver Processing	53
4.2.1.2	CHARPROC	54
4.2.1.3	CANONPROC	55
4.2.2	Write-Side Processing	55
4.2.3	Analysis	55
5	STREAMS Messages	57
5.1	STREAMS Messages Overview	57
5.1.1	Message Types	57
5.1.2	Expedited Data	58
5.2	STREAMS Message Structure	58
5.2.1	Message Linkage	59
5.2.2	Sending and Receiving Messages	61
5.2.3	Control of Stream Head Processing	63
5.2.3.1	Read Options	63
5.2.3.2	Write Options	64
5.3	STREAMS Message Queues and Priority	64
5.3.1	The <code>queue</code> Structure	67
5.3.1.1	Using <code>queue</code> Information	68
5.3.1.2	Queue Flags	68
5.3.1.3	The <code>qband</code> Structure	69
5.3.1.4	Using <code>qband</code> Information	69
5.3.2	Message Processing	70
5.3.2.1	Flow Control	72
5.4	STREAMS Service Interfaces	75
5.4.1	Service Interface Benefits	76
5.4.2	Service Interface Library Example	78
5.4.2.1	Accessing the Service Provider	80
5.4.2.2	Closing the Service Provider	82
5.4.2.3	Sending Data to the Service Provider	82
5.4.2.4	Receiving Data	83
5.4.2.5	Module Service Interface Example	84
5.5	STREAMS Message Allocation and Freeing	88
5.5.0.1	Recovering From No Buffers	90
5.6	STREAMS Extended Buffers	92

6	Polling and Signalling	95
6.1	STREAMS Input and Output Polling	95
6.1.1	Synchronous Input and Output	95
6.1.2	Asynchronous Input and Output	98
6.1.3	Signals	99
6.1.3.1	Extended Signals	100
6.2	STREAMS Stream as Controlling Terminal	100
6.2.1	Job Control	100
6.2.2	Allocation and Deallocation	102
6.2.3	Hung-up Streams	103
6.2.4	Hangup Signals	103
6.2.5	Accessing the Controlling Terminal	103
7	Overview of STREAMS Modules and Drivers	105
7.1	STREAMS Module and Driver Environment	105
7.1.1	Module and Driver Declarations	105
7.1.1.1	Null Module Example	107
7.2	STREAMS Input and Output Controls	109
7.2.1	General <code>ioctl</code> Processing	110
7.2.2	<code>I_STR</code> <code>ioctl</code> Processing	111
7.2.3	Transparent <code>ioctl</code> Processing	112
7.2.4	Transparent <code>ioctl</code> Messages	114
7.2.5	Transparent <code>ioctl</code> Examples	114
7.2.5.1	<code>M_COPYIN</code> Example	114
7.2.5.2	<code>M_COPYOUT</code> Example	117
7.2.5.3	Bidirectional Transfer Example	119
7.2.6	<code>I_LIST</code> <code>ioctl</code>	123
7.3	STREAMS Flush Handling	124
7.4	STREAMS Driver-Kernel Interface	128
7.4.1	Device Driver Interface and Driver-Kernel Interface	130
7.4.2	<i>STREAMS</i> Interface	130
7.5	STREAMS Design Guidelines	131
7.5.1	Module and Driver Rules	131
7.5.1.1	Rules for Open and Close Routines	131
7.5.1.2	Rules for Input Output Controls	132
7.5.1.3	Rules for Put and Service Procedures	132
7.5.2	STREAMS Data Structures	134
7.5.2.1	Dynamic Allocation of STREAMS Data Structures ..	134
7.5.3	Header Files	135
7.5.4	Accessible Symbols and Functions	135
8	STREAMS Modules	139
8.1	Modules	139
8.1.1	Module Routines	139
8.1.2	Filter Module Example	141
8.2	Module Flow Control	144
8.3	Module Design Guidelines	146

9	STREAMS Drivers	149
9.1	Drivers	149
9.1.1	Overview of Drivers	149
9.1.1.1	Driver Classification	149
9.1.1.2	Driver Configuration	150
9.1.1.3	Writing a Driver	150
9.1.1.4	Major and Minor Device Numbers	151
9.1.2	STREAMS Drivers	152
9.1.2.1	Printer Driver Example	154
9.1.2.2	Driver Flow Control	160
9.2	Cloning	160
9.3	Loop-Around Driver	161
9.4	Driver Design Guidelines	168
10	STREAMS Multiplexing	171
10.1	Multiplexing	171
10.1.1	Building a Multiplexor	172
10.1.2	Dismantling a Multiplexor	177
10.1.3	Routing Data Through a Multiplexor	178
10.2	Connecting and Disconnecting Lower Stream	178
10.2.1	Connecting Lower Streams	179
10.2.2	Disconnection Lower Streams	180
10.3	Multiplexor Construction Example	180
10.4	Multiplexing Driver	183
10.4.1	Upper Write Put Procedure	185
10.4.2	Upper Write Service Procedure	188
10.4.3	Lower Write Service Procedure	188
10.4.4	Lower Read Put Procedure	189
10.5	Persistent Links	190
10.6	Multiplexing Driver Design Guidelines	193
11	STREAMS-based Pipes and FIFOs	195
11.1	Pipes and FIFOs	195
11.1.1	Creating and Opening Pipes and FIFOs	195
11.1.2	Accessing Pipes and FIFOs	196
11.1.2.1	Reading from a Pipe or FIFO	196
11.1.2.2	Writing to a Pipe or FIFO	197
11.1.2.3	Closing a Pipe or FIFO	198
11.2	Flushing Pipes and FIFOs	198
11.3	Named Streams	199
11.3.1	fattach	199
11.3.2	fdetach	200
11.3.3	isastream	200
11.3.4	File Descriptor Passing	201
11.3.5	Named Streams in A Remote Environment	201
11.4	Unique Connections	201

12 STREAMS-based Terminal Subsystem . . . 205

12.1	Terminal Subsystem	205
12.1.1	Line Discipline Module	206
12.1.1.1	Default Settings	207
12.1.1.2	Data Structure	207
12.1.1.3	Open and Close Routines	208
12.1.1.4	Read-Side Processing	208
12.1.1.5	Write-Side Processing	209
12.1.1.6	EUC Handling in ldterm	210
12.1.2	Support of termiox(7)	212
12.1.3	Hardware Emulation Module	213
12.2	Pseudo-Terminal Subsystem	214
12.2.1	Line Discipline Module	215
12.2.2	<i>Pseudo-tty</i> Emulation Module PTEM	215
12.2.2.1	Data Structure	217
12.2.2.2	Open and Close Routines	217
12.2.3	Remote Mode	218
12.2.4	Packet Mode	218
12.2.5	Pseudo-tty Drivers ptm and pts	219
12.2.5.1	grantpt	221
12.2.5.2	unlockpt	221
12.2.5.3	ptsname	221

13 Reference 223

13.1	Files	223
13.1.1	User Header Files	223
	STREAMS	223
	STREAMS logger	223
	STREAMS Administrative Driver	223
13.1.2	System Header Files	223
	STREAMS	223
	DDI/DKI	223
	Miscellaneous	224
13.2	Modules	225
13.2.1	Stream Head Module ("sth")	225
13.2.2	Connect Line Discipline Module ("connld")	225
13.2.3	Pipe Module ("pipemod")	225
13.2.4	STREAMS Configuration Module ("sc")	225
13.3	Drivers	225
13.3.1	Clone Driver ("clone")	225
13.3.2	Echo Driver ("echo")	225
13.3.3	FIFO Driver ("fifo")	225
13.3.4	Log Driver ("log")	225
13.3.5	Named STREAMS Device Driver ("nsdev")	225
13.3.6	Null STREAM Driver ("nuls")	225
13.3.7	Pipe Driver ("pipe")	225
13.3.8	STREAMS Administrative Driver ("sad")	225
13.4	System Calls	225

13.4.1	New System Calls	225
13.4.2	Modifications to Old System Calls	226
13.5	Input-Output Controls	226
13.6	Module entry points	227
13.7	Structures	227
13.7.1	STREAMS Structures	227
	Driver Structures	227
	Module Structures	227
	Stream Structures	228
	Queue Structures	228
	Message Structures	228
	Ancillary Structures	228
	Additional Structures	228
13.8	Registration	228
13.8.1	Linux Fast-STREAMS Registration	228
	Registration	228
	Autopush	229
	Administration	229
13.9	Message Handling	229
13.9.1	STREAMS Message Handling Utilities	229
13.10	Queue Handling	230
13.10.1	UP Queue Handling Functions	230
13.10.2	MP Queue Handling Functions	230
13.11	Miscellaneous Functions	231
13.11.1	Miscellaneous DDI/DKI Functions	231
	Memory Functions	231
	Data Movement and Comparison	231
	Device Numbers	231
	Timers	231
	Time, Process and Privilege	232
	Error Logging	232
	File Manipulation	232
13.12	Extensions	232
13.12.1	Common Extensions	232
13.12.2	Linux Fast-STREAMS Extensions	232
	Internal Queue Functions	232
	Flow Control	233
13.12.3	Extensions from LiS 2.18.1	233
13.13	Compatibility	233
13.13.1	SVR 4.2 MP DDI/DKI Compatibility Functions	233
	Atomic Integers	233
	Basic Locks	234
	STREAMS Locks	234
	Read/Write Locks	234
	Priority Levels	235
	Sleep Locks	235
	Synchronization Variables	235
	Resource Allocation	235

13.13.2	AIX 5L Version 5.1 Compatibility Functions	236
13.13.3	HP-UX 11.0i v2 Compatibility Functions	238
13.13.4	OSF/1 1.2/Digital UNIX Compatibility Functions	240
13.13.5	UnixWare 7.1.3 (OpenUnix 8) Compatibility Functions	240
13.13.6	Solaris 9/SunOS 5.9 Compatibility Functions	241
13.13.7	LiS 2.18.1 Compatibility Functions	242
14	Examples	245
14.1	Module Example	245
14.2	Driver Example	245
15	Device Numbers	247
15.1	External Device Numbers	247
15.2	Internal Device Numbers	247
15.3	Clone Device	248
15.3.1	Traditional Cloning	248
15.3.2	New Cloning	248
15.4	Named STREAMS Device	249
15.5	spec File System	249
16	Multi-Threading	251
16.1	Configuration	251
16.2	Synchronous Entry Points	253
16.3	Synchronous Callbacks	254
16.4	Synchronous Callouts	254
16.5	Asynchronous Entry Points	254
16.6	Asynchronous Callbacks	254
16.7	Asynchronous Callouts	254
17	Administration	255
17.1	Administrative Utilities	255
17.1.1	autopush(8)	256
17.1.2	fattach(8)	257
17.1.3	fdetach(8)	258
17.1.4	insf(8)	259
17.1.5	scls(8)	260
17.1.6	strace(8)	261
17.1.7	strclean(8)	262
17.1.8	strconf(8)	263
17.1.9	streams_mknod(8)	264
17.1.10	strerr(8)	265
17.1.11	strinfo(8)	266
17.1.12	strload(8)	267
17.1.13	strsetup(8)	269
17.1.14	strvf(8)	270
17.2	System Controls	271
17.3	/proc File System	273

Appendix A STREAMS Data Structures ... 275

A.1	Stream Structures	275
A.1.1	streamtab	275
A.2	Queue Structures	275
A.2.1	queue	275
A.2.2	qinit	276
A.2.3	module_info	276
A.2.4	module_stat	276
A.2.5	qband	276
A.3	Message Structures	277
A.3.1	msgb	277
A.3.2	datab	278
A.4	Input Output Control Structures	278
A.4.1	iocblk	278
A.4.2	copyreq	278
A.4.3	copyresp	279
A.4.4	striocctl	279
A.5	Link Structures	279
A.5.1	linkblk	279
A.6	Options Structures	280
A.6.1	stroptions	280

Appendix B STREAMS Message Types 281

B.1	Message Types	281
B.2	Ordinary Messages	281
B.3	High Priority Messages	289

Appendix C STREAMS Utilities 295

C.1	Utility Descriptions	295
C.1.1	adjmsg-trim bytes in a message	296
C.1.2	allocb-allocate a message and data block	296
C.1.3	backq-get pointer to the queue behind a given queue	296
C.1.4	bcnput-test for flow control in the given priority	296
C.1.5	bufcall-recover from failure of allocb	297
C.1.6	canput-test for room in a queue	297
C.1.7	copyb-copy a message block	297
C.1.8	copymsg-copy a message	298
C.1.9	datamsg-test whether message is a data message	298
C.1.10	dupb-duplicate a message block descriptor	298
C.1.11	dupmsg-duplicate a message	298
C.1.12	enableok-re-allow a queue to be scheduled for service...	299
C.1.13	esballoc-allocate message and data blocks	299
C.1.14	flushband-flush the messages in a given priority band ..	299
C.1.15	flushq-flush a queue	299
C.1.16	freeb-free a single message block	300
C.1.17	freemsg-free all message blocks in a message	300
C.1.18	getadmin()-return the pointer to the module	300

C.1.19	getmid-return a module id	300
C.1.20	getq-get a message from a queue	300
C.1.21	insq-put a message at a specific place in a queue	301
C.1.22	linkb-concatenate two messages into one	301
C.1.23	msgdsize-get the number of data byptes in a message ..	301
C.1.24	noenable-prevent a queue from being scheduled	301
C.1.25	OTHERQ-get pointer to the mate queue	302
C.1.26	pullupmsg-concatenate and align bytes in a message ...	302
C.1.27	putbq-return a message to the beginning of a queue....	302
C.1.28	putctl-put a control message	302
C.1.29	putctl1-put a control message with a one-byte parameter	303
C.1.30	putnext-put a message to the next queue.....	303
C.1.31	putq-puta message on a queue.....	303
C.1.32	qenable-enable a queue	304
C.1.33	qreply-send a message on a <i>Stream</i> in the reverse direction	304
C.1.34	qsize-find the number of messages on a queue.....	304
C.1.35	RD-get pointer to the read queue.....	304
C.1.36	rmvb-remove a message block from a message	304
C.1.37	rmvq-remove a message from a queue	305
C.1.38	splstr-set processor level	305
C.1.39	strlog-submit messages for logging	305
C.1.40	strqget-obtain information about a queue or band of the queue	306
C.1.41	strqset-change information about a queue or band of the queue	306
C.1.42	testb-check for an available buffer	307
C.1.43	unbufcall-cancel a bufcall request	307
C.1.44	unlinkb-remove a message block from the head of a message	307
C.1.45	WR-get pointer to the write queue	307
C.2	DKI Interface	307
C.3	Utility Routine Summary.....	307

Appendix D STREAMS Debugging 309

D.1	Debugging	309
D.2	crash(1M) Command	310
D.3	Dump Module Example	312
D.4	Error and Trace Logging	320

Appendix E STREAMS Configuration 323

E.1	Configuration.....	323
E.1.1	Configuring STREAMS Modules and Drivers	323
E.1.1.1	Configuration Examples	324
E.1.1.2	Tunable Parameters.....	325
E.1.2	Autopush Facility	326
E.1.2.1	User Interface	326

Appendix F Conformance 329

F.1	SVR 4.2 MP DDI/DKI Compatibility	329
F.2	AIX 5L Version 5.1 Compatibility	329
F.3	HP-UX 11.0i v2 Compatibility	329
F.4	OSF/1 1.2/Digital UNIX Compatibility	329
F.5	UnixWare 7.1.3 Compatibility	329
F.6	Solaris 9/SunOS 5.9 Compatibility	329
F.7	Super/UX Compatibility	329
F.8	UXP/V Compatibility	329
F.9	LiS 2.18.1 Compatibility	329

Appendix G Portability 331

G.1	Porting with Core Function Support	331
G.1.1	Core Message Functions	331
G.1.2	Core UP Queue Functions	331
G.1.3	Core MP Queue Functions	332
G.1.4	Core DDI/DKI Functions	332
G.1.5	Some Common Extension Functions	333
G.1.6	Some Internal Functions	333
G.1.7	Some Oddball Functions	333
G.2	Porting from SVR 4.2 MP	334
G.2.1	Differences from SVR 4.2 MP	334
G.2.2	Commonalities with SVR 4.2 MP	334
G.2.3	Compatibility functions for SVR 4.2 MP	334
G.2.3.1	Priority Levels	334
G.2.3.2	Atomic Integers	335
G.2.3.3	Basic Locks	336
G.2.3.4	STREAMS Locks	336
G.2.3.5	Read/Write Locks	336
G.2.3.6	Sleep Locks	336
G.2.3.7	Synchronization Variables	336
G.2.3.8	Resource Allocation	336
G.2.3.9	Device Numbering	337
G.2.4	Configuration ala SVR 4.2 MP	337
G.3	Porting from AIX 5L Version 5.1	337
G.3.1	Differences from AIX 5L Version 5.1	337
G.3.2	Commonalities with AIX 5L Version 5.1	337
G.3.3	Compatibility Functions for AIX 5L Version 5.1	337
G.3.3.1	Core Extensions	337
G.3.3.2	Common Module Utilities	337
G.3.3.3	Registration	337
G.3.3.4	Message Filtering	338
G.3.4	Configuration ala AIX 5L Version 5.1	338
G.4	Porting from HP-UX 11.0i v2	338
G.4.1	Differences from HP-UX 11.0i v2	338
G.4.2	Commonalities with HP-UX 11.0i v2	338
G.4.3	Compatibility Functions for HP-UX 11.0i v2	338
G.4.3.1	Core Extensions	338

G.4.3.2	Registration	338
G.4.3.3	Sleeping	338
G.4.4	Configuration ala HP-UX 11.0i v2	338
G.5	Porting from OSF/1 1.2/Digital UNIX	338
G.5.1	Differences from OSF/1 1.2/Digital UNIX	339
G.5.2	Commonalities with OSF/1 1.2/Digital UNIX	339
G.5.3	Compatibility Functions for OSF/1 1.2/Digital UNIX ...	339
G.5.3.1	Core Extensions	339
G.5.3.2	Common Module Utilities	339
G.5.3.3	Registration	339
G.5.3.4	Others	339
G.5.4	Configuration ala OSF/1 1.2/Digital UNIX	339
G.6	Porting from UnixWare 7.1.3 (OpenUnix 8)	339
G.6.1	Differences from UnixWare 7.1.3 (OpenUnix 8)	339
G.6.2	Commonalities with UnixWare 7.1.3 (OpenUnix 8)	339
G.6.3	Compatibility Functions for UnixWare 7.1.3 (OpenUnix 8)	
	339
G.6.3.1	Device Numbering	339
G.6.3.2	Memory Alignment	340
G.6.3.3	Direct <i>STREAMS</i> Input-Output Controls	340
G.6.4	Configuration ala UnixWare 7.1.3 (OpenUnix 8)	340
G.7	Porting from Solaris 9/SunOS 5.9	340
G.7.1	Differences from Solaris 9/SunOS 5.9	340
G.7.2	Commonalities with Solaris 9/SunOS 5.9	340
G.7.3	Compatibility Functions for Solaris 9/SunOS 5.9	340
G.7.3.1	<i>STREAMS</i> Queue Referenced Callbacks	341
G.7.3.2	<i>STREAMS</i> Registration	341
G.7.3.3	DDI	341
G.7.3.4	Loadable Module Interface	341
G.7.4	Configuration ala Solaris 9/SunOS 5.9	342
G.8	Porting from Super/UX	342
G.8.1	Differences from Super/UX	342
G.8.2	Commonalities with Super/UX	342
G.8.3	Compatibility Functions for Super/UX	342
G.8.4	Configuration ala Super/UX	342
G.9	Porting from UXP/V	342
G.9.1	Differences from UXP/V	342
G.9.2	Commonalities with UXP/V	342
G.9.3	Compatibility Functions for UXP/V	342
G.9.4	Configuration ala UXP/V	342
G.10	Porting from LiS 2.18.1	342
G.10.1	Differences from LiS 2.18.1	342
G.10.2	Commonalities with LiS 2.18.1	342
G.10.3	Compatibility Functions for LiS 2.18.1	342
G.10.3.1	Extensions	342
G.10.3.2	Device Creation and Deletion	343
G.10.3.3	Registration	343
G.10.4	Configuration ala LiS 2.18.1	343

G.11	Developing Portable STREAMS Modules	343
G.11.1	Memory Allocation	343
G.11.2	Alignment of Message Buffers	344
G.11.3	Disabling and Enabling Queue Procedures	344
G.11.4	Freezing and Unfreezing Streams	344
G.11.5	Passing Messages from Interrupt Service Routines	344
G.11.6	Timeout Call Back and Link Identifiers	344
G.11.7	Synchronization with Timeouts and Callback Functions	344
G.11.8	Synchronization with Callout Functions	344
G.11.9	Synchronization of Drivers and Modules	344
G.11.10	Special <i>STREAMS</i> Message Types	344
G.11.11	Use of Message Allocation Priorities	344
G.11.12	Registration and Deregistration	344
G.11.13	Device Numbering	344
Appendix H	Copying	347
H.1	GNU General Public License	347
H.1.1	Preamble	347
H.1.2	Terms and Conditions for Copying, Distribution and Modification	348
H.1.3	How to Apply These Terms to Your New Programs	352
H.2	GNU Free Documentation License	353
H.2.1	Preamble	353
H.2.2	Terms and Conditions for Copying, Distribution and Modification	353
H.2.3	How to use this License for your documents	359
Index		361

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

1 Introduction

The [OpensS7 Project](#), ‘streams-0.7a.3’ package provides an *SVR 4.2 MP* compatible *STREAMS* implementation for **Linux** 2.4 and 2.6 series kernels.

1.1 What is *STREAMS*?

STREAMS is a facility first presented 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

1.2 Does Linux have *STREAMS*?

No, not as part of the kernel. That is rather peculiar, particularly since Linux normally follows *SVR4* first and 4BSD second. (Otherwise, it would just be another BSD.)

A number of attempts were made to move the *Linux STREAM (LiS)* project into the **Linux** kernel, however, each attempt crashed and burned in a shower of flames from BSD advocates on LKML. Arguments against appear to be based more on *Religious Denomination* rather than valid technical argument. This is discussed more in the next section (see [Section 1.3 \[Why *STREAMS*?\]](#), page 4).

1.2.1 Linux *STREAMS* (LiS)

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

1.2.2 Linux Fast-STREAMS (LfS)

1.2.3 STREAMS for FreeBSD

1.2.4 OpenSTREAMS?

1.3 Why STREAMS?

1.3.1 STREAMS versus Sockets

The basic question that is always asked is: "Why use *STREAMS* when you can just use **Linux**'s NET4 BSD Sockets instead?"

The answer to this question is that *STREAMS* provides capabilities for specialized protocols and streamed input/output requirements (such as media) that are not amenable to the sockets interface or queue mechanisms.

Two examples are SS7 (Signalling System Number 7) which is a specialized Telecommunications protocol used by switching equipment in the Public Switched Telephone Network; and transferring and manipulating voice channels associated with telephone call or other telecommunications services. These are the reasons why the **OpenSS7 Project** originally embarked on using *STREAMS*. You will find that a large number of SS7 stack vendors also deliver *UNIX* and even *RTOS* products on *STREAMS*.

Although the BSD Sockets framework was established to permit arbitrary protocols to be implemented within the framework, it is seldom that BSD Sockets is actually used in this fashion.

The 4.2BSD version of *UNIX* introduced *sockets* [Leffler, McKusick, Karels, Quateman 1988]. The operating system provided an infrastructure in which network protocols could be implemented. It provided memory management facilities, a set of system calls for accessing network software, an object-oriented framework for the network protocols themselves, and a formalized device driver interface. The *sockets* mechanism was primarily used to implement the TCP/IP protocols for the ARPA Internet. The device driver interface made it possible for the operating system to support a wide range of network controllers. *sockets* are widely used for the implementation of TCP/IP on *UNIX* systems and have been ported to many implementations of *UNIX System V*. Although it is possible to implement other protocols within the *sockets* mechanism, it was not often done.

An alternative infrastructure for providing network protocols is *STREAMS*, originally designed by Dennis Ritchie [Ritchie 1984a] and first released in *UNIX System V Release 3.0*. *STREAMS* provides an environment in which communications protocols can be developed. It consists of a set of system calls, kernel functions and data structures. With this environment it is easier to write modular and reusable code. The code is also simpler because many of the support functions the programmer needs are provided by the *STREAMS* infrastructure.

Have you ever seen the *RTP* (*Real-Time Transport Protocol*, *RFC 1889*) implemented under a Socket? Why not? Is not the sockets interface so flexible as to permit such protocols to be implemented?

There are several reasons that *BSD Sockets* have not been used for other protocol development:

- Although *BSD Sockets* provides a framework for protocol development, it does not provide many utility functions for working with arbitrary protocols. Most of the utilities are DARPA ARPANET specific.⁴
- Protocol to protocol module interfaces are poorly standardized for the *BSD Sockets* system, whereas, protocol to protocol module service interfaces are well defined under OSI for *STREAMS*. (**Linux** discards the protocol to protocol interface anyway.)
- The *BSD Sockets* interface can easily be applied over *STREAMS* transport protocol modules; however, the reverse is not true: the *STREAMS* interface cannot easily be provided over the *BSD Sockets* protocol modules.⁵
- Support in the *BSD Sockets* model for dynamically loaded protocol (kernel) modules and administrative reconfiguration of protocols and interfaces for new protocols are poorly supported.
- The *BSD Sockets* model has almost no support for banded or priority message queues within the model and no systemic approach to flow control.
- The **Linux** implementation of *BSD Sockets* discards much of the general purpose protocol framework, presumably in the pursuit of speed.

BSD sockets consists of a socket that interfaces with the user using file system and socket semantics, a protocol control block that represents the upper-most protocol, a socket to protocol interface, additional protocol control blocks representing lower protocol components, a protocol to protocol interface, a device interface abstraction, and a protocol to device interface.

Linux discards the protocol control block, socket to protocol interface, protocol to protocol interface, and protocol to device interface.

One of the major ramifications of **Linux** discarding the protocol to protocol interface is that it is very difficult to implement layered or tunnelled protocols in the **Linux** kernel.⁶ Layered protocols that run, say, over UDP, such as econet, must internally use the socket interface to a UDP datagram socket to layer the econet protocol over UDP. Under *STREAMS* it is much easier to either push econet as a module over the UDP transport provider stream, or to I_LINK transport provider streams under an econet multiplexing driver.

In commenting on the relative performance of *STREAMS* and Sockets, Mitchel Waite had this to say:

⁴ This should not be suprising as the 4BSD releases were developed for DARPA.

⁵ A case in point is the iBCS. You will see in the iBCS that, although a basic XTI over Sockets implementation can be provided, none of the *STREAMS* facilities can be supported. In constrast the *STREAMS* INET driver that performs XTI over Sockets with in the *STREAMS* framework is easily implemented as a single device driver and provides both *iBCS* and *STREAMS* capabilities.

⁶ That is, even more difficult than on a BSD system.

Sockets are like pipes with more power. They are bidirectional and may cross network or other machine boundaries. In addition, sockets allow limited control information as well as data.

Streams are more general still, with extensive control information passing capabilities.

On most *UNIX* systems, messages (if available) have the lowest overhead and highest bandwidth, with pipes following close behind. Because they support complex networking facilities, sockets are probably less efficient than streams, but because they rarely appear on the same machine as streams, the question is somewhat academic. They certainly have much lower bandwidth than pipes or messages.⁷

With *Linux Fast-STREAMS* it will be very possible to compare the performance of *STREAMS* in comparison to *Sockets*. It will also be possible to compare the performance of traditional **Linux** pipes and FIFOs with *STREAMS*-based pipes and FIFOs.

1.3.2 Benefits of *STREAMS*

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

STREAMS also greatly simplifies the user interface for languages that have complex input and output requirements.

1.3.2.1 Standardized Service Interfaces

STREAMS simplifies the creation of modules that present a service interface to any neighbouring application program, module, or device driver. A service interface is defined at the boundary between two neighbours. In *STREAMS*, a *service interface* is a set of messages and the rules that allow passage of these messages across the boundary. A module that implements a service interface will receive a message from a neighbour and respond with an appropriate action (for example, send back a requires to retransmit) based on the specific message received and the preceding sequence of messages.

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

1.3.2.2 Manipulating Modules

STREAMS provides the ability to manipulate modules from user level, to interchange modules with common service interfaces, and to change the service interface to a *STREAMS* user

⁷ *UNIX Papers, for UNIX Developers and Power Users*, (Waite, 1987) pp. 358-359

process. These capabilities yield further benefits when implementing networking services and protocols, including:

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

1.3.2.3 Protocol Portability

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

1.3.2.4 Protocol Substitution

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

1.3.2.5 Protocol Migration

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

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

1.3.2.6 Module Reusability

Figure 15 shows the same canonical module (for example, one that provide 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*.

1.3.3 Criticism of STREAMS

Following are some excerpts from Dennis M. Ritchie's original (1984) Bell Technical Journal paper on the stream I/O system. These excerpts are the limitations of the system as were perceived by Dennis M. Ritchie at the time. Strangely enough, although every limitation listed by Dennis was fixed even as early as *UNIX System V Release 3.0* and even in the

UNIX Eighth Edition, some BSD advocates will use these limitations as a reason for not using *STREAMS* in BSD. Also, note that BSD'ers will also say that *STREAMS* was a *UNIX Eighth Edition* (Bell Laboratories Research version of UNIX) thing; however, Dennis' paper clearly states that the base system for the initial *Stream Input-Output System* was *4.1BSD*. Also note that *4.1BSD* already had *sockets* and that some of Ritchie's work was taken from *sockets*. It took until *4.2BSD* for *BBN* to add the *DARPA* protocol stack to *sockets*.

Perhaps it is not so surprising why *BSD*'ers hark back to Ritchie's original problem list for *STREAMS*: because it was at that point that *BSD* decided to not follow the *STREAMS* work too closely, except as regards IPC, and *UNIX* domain mechanisms. It is likely that *BSD* would have used *STREAMS*, however, it was included in *UNIX System V Release 3.0* and this was the first release that *AT&T* was allowed to aggressively market under the terms of the *Modified Judgement*.

Although the new organization performs well, it has several peculiarities and limitations. Some of them seem inherent, some are fixable, and some are the subject of current work.

I/O control calls turn into messages that require answers before a result can be returned to the user. Sometimes the message ultimately goes to another user-level process that may reply tardily or never. The stream is write-locked until the reply returns, in order to eliminate the need to determine which process gets which reply. A timeout breaks the lock, so there is an unjustified error return if a reply is late, and a long lockup period if one is lost. The problem can be ameliorated by working harder on it, but it typifies the difficulties that turn up when direct calls are replaced by message-passing schemes.

This problem was never really fixed I suppose because most *STREAMS* specifications say that only one *ioctl* can be outstanding for a given stream. Nevertheless, an *ioctl* identifier was added to the *M_IOCTL* message that uniquely identifies the *ioctl*; but, a timer is still used. With the *I_STR* *ioctl*, however, the caller has control over the duration of the timeout. Strange, but this unfixed problem is the one that seldom gets raised as a reason for not using *STREAMS*.

Several oddities appear because time spent in server routines cannot be assigned to any particular user or process. It is impossible, for example, for devices to support privileged *ioctl* calls, because the device has no idea who generated the message. Accounting and scheduling becomes less accurate; a short census of several systems showed that between 4 and 8 per cent of non-idle CPU time was being spent in server routines. Finally, the anonymity for server processing most certainly makes it more difficult to measure the performance of the new I/O system.

This problem with privileged *ioctl* calls was easily fixed by adding the credentials of the caller to the *M_IOCTL* message. This limitation is also not mentioned by *STREAMS* critics.

In its current form the stream I/O system is purely data-driven. That is, data is presented by a user's *write* call, and passes through to the device; conversely, data appears unbidden from a device and passes to the top level, where it is picked up by *read* calls. Wherever possible flow control throttles down fast

generators of data, but nowhere except at the consumer end of a stream is there knowledge for precisely how much data is desired. Consider a command to execute possibly interactive program on another machine connected to a stream. The simplest such common sets up the connection and invokes the remote program, and then copies characters from its own standard input to the stream, and from the stream to its standard output. The scheme is adequate in practise, but breaks when the user types more than the remote program expects. For example, if the remote program reads no input at all, any typed-ahead characters are sent to the remote system and lost. This demonstrates a problem, but I know of no solution inside the stream I/O mechanism itself; other ideas will have to be applied.

Back-enabling of queues and the use of the `M_READ` message makes it possible for the consumer end of the stream to signal its desire for data downstream. Also, in the example that Ritchie gives here, the network protocol (TCP) is of no help either. This limitation is also not mentioned by *STREAMS* critics.

Streams are linear connections; by themselves, they support no notion of multiplexing, fan-in or fan-out. Except at the ends of a stream, each invocation of a module has a unique "next" and "previous" module. Two locally-important applications of streams testify to the importance of multiplexing: Blit terminal connections, where the multiplexing is done well, though at some performance costs, but a user program, and remote execution of commands over a network, where it is desired, but not now easy, to separate the standard output from error output. It seems likely that a general multiplexing mechanism could help in both cases, but again, I do not yet know how to design it.

This was, of course, solved, even in *UNIX System V Release 3.0* with the `I_LINK`, `I_PLINK`, `I_UNLINK` and `I_PUNLINK` *STREAMS* `ioctl` commands and the concept of a multiplexing pseudo-device driver. This fixed limitation you will see mentioned below.

The following excerpt shows how *BSD*'ers like to misinterpret the situation:

Original work on the flexible configuration of IPC processing modules was done at Bell Laboratories in *UNIX Eighth Edition* [Presotto & Richie, 1985]. This *stream I/O system* was based on *UNIX* character I/O system. It allowed a user process to open a raw terminal port and then to insert appropriate kernel-processing modules, such as one to do normal terminal line editing. Modules to process network protocols also could be inserted. Stacking a terminal-processing module on top of a network-processing module allowed flexible and efficient implementation of *network virtual terminals* within the kernel. A problem with streams modules, however, is that they are inherently linear in nature, and thus they do not adequately handle the fan-in and fan-out associated with multiplexing in datagram-based networks; such multiplexing is done in device drivers, below the modules proper. The Eighth Edition stream I/O system was adopted in System V, Release 3 as the *STREAMS* system.⁸

⁸ *The Design and Implementation of the 4.4BSD Operating System*, McKusick, et. al., (Addison-Wesley, 1996) pp. 15-16

Well, the *UNIX Eighth Edition Stream Input-Output System* may have been included in *UNIX System V Release 3.0* as stated, however, Ritchie's *Stream Input-Output System* was implemented on *4.1BSD* for the October 1984 paper.

The design of the networking facilities for *4.2BSD* took a different approach, based on the *socket* interface and a flexible multilayer network architecture. This design allows a single system to support multiple sets of networking protocols with stream datagram, and other types of access. Protocol modules may deal with multiplexing of data from different connection onto a single transport medium, as well as with demultiplexing of data for different protocols and connection received from each network device. The *4.4BSD* release made small extensions to the *socket* interface to allow the implementation of the *ISO* networking protocols.⁹

1.3.4 Realities of STREAMS

The realities of *STREAMS* are as follows:

- *STREAMS* is implemented on every major “Big Iron” *UNIX*.

Even Sun Microsystems chose to abandon *BSD Sockets* as an internal kernel networking implementation and moved to the *UNIX System V Release 4 STREAMS* subsystem instead.¹⁰ *STREAMS* is implemented (to list a few) in *AIX 5L Version 5.1 PSE*, *HP-UX 11.0i v2 STREAMS/UX*, *OSF/1 1.2/Digital UNIX*, *UnixWare 7.1.3 (OpenUnix 8)*, *Solaris 9/SunOS 5.9*, *Super-UX*, *UXP/V* and *MacOS OT*.

- *STREAMS* is implemented on many Real-Time Operating Systems (RTOS) based on *UNIX*.

Examples include *WindRiver*, *PSOS*, *VxWorks*, etc.

- *STREAMS* implementations are widely standardized on the *UNIX System V Release 4.2* specifications.
- *STREAMS* provides standardized (POSIX, OpenGroup, SVID) Transport Library Interface to communications networking suites.
- *STREAMS* has many portable implementations.

1.4 Why Fast?

Linux Fast-STREAMS includes the word *fast* in the name because of the original roots of the *Linux Fast-STREAMS* development effort. *Linux Fast-STREAMS* was originally developed by the **OpenSS7 Project** as a production replacement for the *Linux STREAMS (LiS)* package previously available from **GCOM**. One of the reasons for contemplating a replacement for *Linux STREAMS (LiS)* was the dismal performance provided by *Linux STREAMS (LiS)*. Other reasons included:

⁹ *The Design and Implementation of the 4.4BSD Operating System*, McKusick, et. al., (Addison-Wesley, 1996) pp. 15-16

¹⁰ BSD'er will tell you that Sun Microsystems just made a bad decision.

- Mainline Adoption instead of Portability

LiS attempts to maintain portability across a number of operating systems. The goals of portability and mainline adoption are usually at cross-purposes. *Linux Fast-STREAMS* proposes mainline adoption in contrast to portability. Many *STREAMS* implementations are available for other operating systems.

- Production Grade

LiS attempts to always provide debugging facilities¹¹ and does not trust the driver or module writer. This leads to poor performance and in many cases the propagation of bugs to the field by failing to panic the kernel. *Linux Fast-STREAMS* aims at a production grade environment that implicitly trusts the driver or module while providing optional debugging facilities (both compile-time options as well as run-time options).

- SVR 4.2 MP Compatibility

LiS only provides *SVR 4* uniprocessor capabilities. *Linux Fast-STREAMS* provides *SVR 4.2 MP* capabilities.

- Portability

LiS forces ported drivers and modules from other implementations to use the *LiS* DDI and configuration mechanisms. *Linux Fast-STREAMS* provides compatibility functions for all major implementations of *STREAMS* as well as providing a rich DDI based on *SVR 4.2 MP*, *Solaris*, and other implementations.

- Bug Circumvention

- Major Redesign

- Scalable

- Soft Real Time Performance

LiS avoids use of high-performance **Linux**-specific facilities because of its aims at portability. *Linux Fast-STREAMS* being aimed at only **Linux** uses the highest-performance techniques available in the **Linux** kernel for implementation. This includes kernel memory caches and other techniques.

- Maintainability

1.5 Why Linux?

Well, **Linux** is the only *SVR 4* based system that does not provide *STREAMS*, although *STREAMS* is an essential part of *SVR 4*. Without *STREAMS*, **Linux** is just another *BSD*, and perhaps a bad one.

1.6 Why Compatibility?

Linux Fast-STREAMS is designed and implemented to be compatible with as many *SVR 4.2 MP* based implementations of *STREAMS* as possible. This is done for several reasons:

¹¹ In fact, these debugging facilities always point at the driver or module writer when a bug is encountered in *LiS* itself!

1. *Porting legacy drivers to **Linux**:*

Many legacy *STREAMS* drivers have been written and developed for *SVR 4.2 MP* or *UNIX* systems based on *SVR 4.2 MP*. Remaining compatible with as many implementation as possible permits these legacy drivers to be easily ported from their native *UNIX* variant to the *Linux Fast-STREAMS* environment, thus quickly porting these legacy drivers to **Linux**.

2. *Leverage of knowledge base:*

Many developers are familiar one or another of the mainstream *UNIX* implementations of *SVR 4.2 MP STREAMS*. By remaining as compatible as possible with all these implementations of *STREAMS* permits knowledge and expertise in the *UNIX* variant of *STREAMS* to be transferred and applied to *Linux Fast-STREAMS* on **Linux**.

3. *Reverse portability:*

Because it is as compatible as possible with other *STREAMS* implementations, *STREAMS* drivers and modules developed on *Linux Fast-STREAMS* can easily be ported to other implementations if a set of compatibility and portability guidelines are followed. This allows *STREAMS* drivers and modules developed on the **Linux** operating system to be used on branded *UNIX* systems with minimal porting and modification.

4. *Standardization:*

By being as compatible as possible with as many *STREAMS* implementations as possible, *Linux Fast-STREAMS* implements an *ipso facto* standard. Unfortunately, the *OpenGroup* and *POSIX* have been very lacking in the standardization of internal kernel interfaces such as *STREAMS*. Maximum compatibility moves close to providing a standard for such interfaces.

1.6.1 Intel Binary Compatibility Suite (iBCS)

The *Intel Binary Compatibility Suite* provides binary compatibility on the *Intel* architecture for systems conforming to *SVR 4.2*. *RedHat* has released an *iBCS* module for their distributions of **Linux** and the **Linux Kernel** for some time.

1.6.1.1 OpenGroup Specifications

OpenGroup and POSIX specifications have never directly addressed *STREAMS* implementation within the operating system. I suppose that this is primarily because 4BSD based system have seldom included *STREAMS*. Perhaps it was due to some religious upheaval from BSD advocates that did not want to see *STREAMS* become part of a standard.

Nevertheless, the *STREAMS* subsystem has been an optional part of the **OpenGroup** specifications for some time. It is my opinion that the **OpenGroup** has missed a rich opportunity for standardization of kernel level interfaces.

UNIX 03 Compliance

UNIX 03 compliance to *Open Group Extensions* requires that XTI/TLI networking support be provided. (See *XNS 5.2*). As the *iBCS* has proven, this does not require full *STREAMS* support, however, it is an easier thing to accomplish with *STREAMS* support. Even though

the *XNS 5.2* specification does not describe *STREAMS*, the *SUSv3* does. The **OpenGroup** has never defined the internals of the *STREAMS* facility in their *CAE* specifications; however, they are described and the user-space facilities and system calls are completely defined and described.

UNIX 98 Compliance

UNIX 98 compliance to *X/Open Extensions* requires that XTI/TLI networking support be provided. (See *XNS 5*). As the *iBCS* has proven, this does not require full *STREAMS* support, however, it is an easier thing to accomplish with *STREAMS* support. Even though the *XNS 5* specification does not describe *STREAMS*, the *XSI 5* and *SUSv2* does. The **OpenGroup** has never defined the internals of the *STREAMS* facility in their *CAE* specifications; however, they are described and the user-space facilities and system calls are completely defined and described.

UNIX 95 Compliance

UNIX 95 compliance to *X/Open Extensions* requires that XTI/TLI networking support be provided. (See *XNS 4.2*). As the *iBCS* has proven, this does not require full *STREAMS* support, however, it is an easier thing to accomplish with *STREAMS* support. Even though the *XNS 4.2* specification does not describe *STREAMS*, the *XSI 4.2* and *SUS* does. The **OpenGroup** has never defined the internals of the *STREAMS* facility in their *CAE* specifications; however, they are described and the user-space facilities and system calls are completely defined and described.

1.6.2 Device Driver Interface (DDI)

2 Overview of Linux Fast-STREAMS

This manual documents how to develop and port STREAMS drivers and modules for Linux Fast-STREAMS.

2.1 *STREAMS* Definitions

Stream A *Stream* is a full-duplex communications path between a User Process and a Kernel Level Device or Pseudo-Device Driver. A *Stream* is a group of *STTREAMS* message queue pairs in a chain from a kernel driver at the *Stream End* to a *Stream Head* at the user.

Stream Head
A *Stream Head* is the component in a *Stream* that is closest to the User Process. The *Stream Head* sits in Kernel Space and provide interface to the User Process.

Stream End
A *Stream End* is the component in a *Stream* that is farthest from the User Process. The *Stream End* sits in Kernel Space and is normally a Device or Pseudo-Device Driver.

Module A *Module* is a protocol module that resides between the *Stream Head* and *Stream End* on a *Stream*. Protocol *Modules* are optional and can be pushed or popped from a *Stream* using `ioctl` commands to the *Stream Head*. A *Module* can refer to a linkable kernel component (kernel module), a *STREAMS* driver, module or mux (*STREAMS* module), or a pushable *STREAMS* module.

Driver A *Driver* is a device or pseudo-device driver that sits at the *Stream End*. *Drivers* are associated with device numbers and are opened by the User Process. A *Driver* is a *STREAMS* module which sets at a *Stream End*, farthest from the user on a *Stream*.

Messages A *Message* is a chain of data buffers used for passing data by reference between neighbouring *STREAMS Modules*, the *Stream Head* and the *Driver*. A *Message* is a message containing data that is made up of a sequence of message blocks, data blocks and data.

Queues Each *Stream Head*, *Stream End* and intervening *Module* consists of a pair of *Queues*: one downstream *Queue* for messages first written by the User Process, and one upstream *Queue* for messages ultimately read by the User Process. A *Queue* is an ordered list of messages awaiting processing. *STREAMS* modules form a full duplex connection by pairing queues as read and write queues.

Stream I/O Multiplexing

Multiplexing is a situation where multiple *Streams* can fan-in or fan-out from a *Module* or *Multiplexing Driver*. A *Multiplexing* driver or *mux* is a *STREAMS* module which has lower as well as upper queue pairs.

Polling

Flow Control

A *Stream* is flow-controlled when it is congested and temporarily cannot pass messages in the band in which the stream is flow-controlled.

2.2 Concepts

There are three types of *STREAMS* entities: the stream head, modules and drivers. Stream heads are the kernel-space interface to the user process. Modules are pushable streams protocol modules that are optionally placed between the user process (stream head) and the driver (stream end). Drivers are device drivers or pseudo device drivers that can be opened using a character device special file.

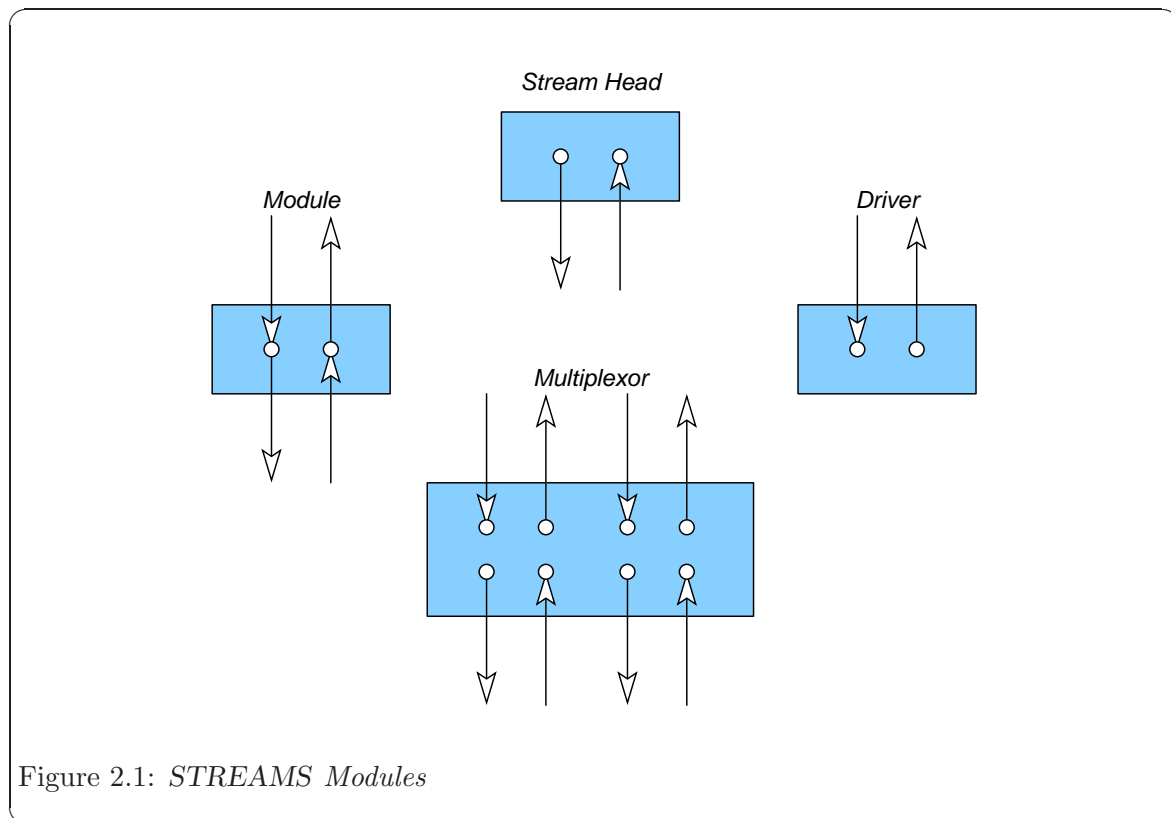


Figure 2.1 illustrates the basic *STREAMS* modules:

Stream Head

The *Stream Head* is allocated by the *STREAMS* subsystem when a *STREAMS* character special device is opened.

A stream head is created whenever the first `open(2)` is performed on a *STREAMS* device. A stream head has a queue pair just like any other driver, module or multiplexer, but it interfaces directly with user-space library calls. A stream head is opened whenever a driver is opened (whether for the first

time or not), and each time that a module is pushed onto the stream. A stream head is closed when the last close is performed on the stream.

Driver *Drivers* are opened by character major and minor device number.

Instances of a *STREAMS* driver are created by calling `open(2)` on a character special device which has a device major number which has been registered against the driver. Opening the special character device results in the driver's open procedure being called. The driver's open procedure is also called each time that the device is opened and each time that a module is pushed onto a stream. Closing the special character device for the last time results in the driver's close procedure being called.

Module *Modules* may be pushed under a *Stream Head* once a *Driver* or *Multiplexer* has been opened.

STREAMS modules are not created by calling `open(2)`: they are pushed onto an open stream with the `I_PUSH ioctl(2)`. Modules are pushed by name. Modules can be popped from a stream using the `I_POP ioctl(2)`. Several modules can be pushed on a stream. A module's open procedure is called when it is pushed and whenever the stream is reopened, and its close procedure is called whenever it is popped, or when the stream is closed for the last time and is being dismantled.

Multiplexer

Multiplexers are opened on the upper interface like *Drivers*, but can also have other streams *linked* underneath them.

STREAMS Multiplexers have both an upper queue pair and a lower queue pair. Upper queue pairs are opened when a character special device is opened as with a driver. Lower queue pairs are linked when another stream is linked under the *Multiplexer* by executing the `I_LINK` or `I_PLINK ioctl(2)` command on a control stream associated with the *Multiplexer*.

`I_LINK`'ed streams can be unlinked with the `I_UNLINK ioctl(2)` command. `I_PLINK`'ed streams can be unlinked with the `I_PUNLINK ioctl(2)` command. When the control stream is closed, all `I_LINK`'ed streams are automatically unlinked. `I_PLINK`'ed streams remain linked until the last upper stream of the *Multiplexer* is closed, at which time the *Multiplexer* is dismantled and the `I_PLINK`'ed streams are `I_PUNLINK`'ed and closed.

A simple stream is illustrated in [Figure 2.2](#).

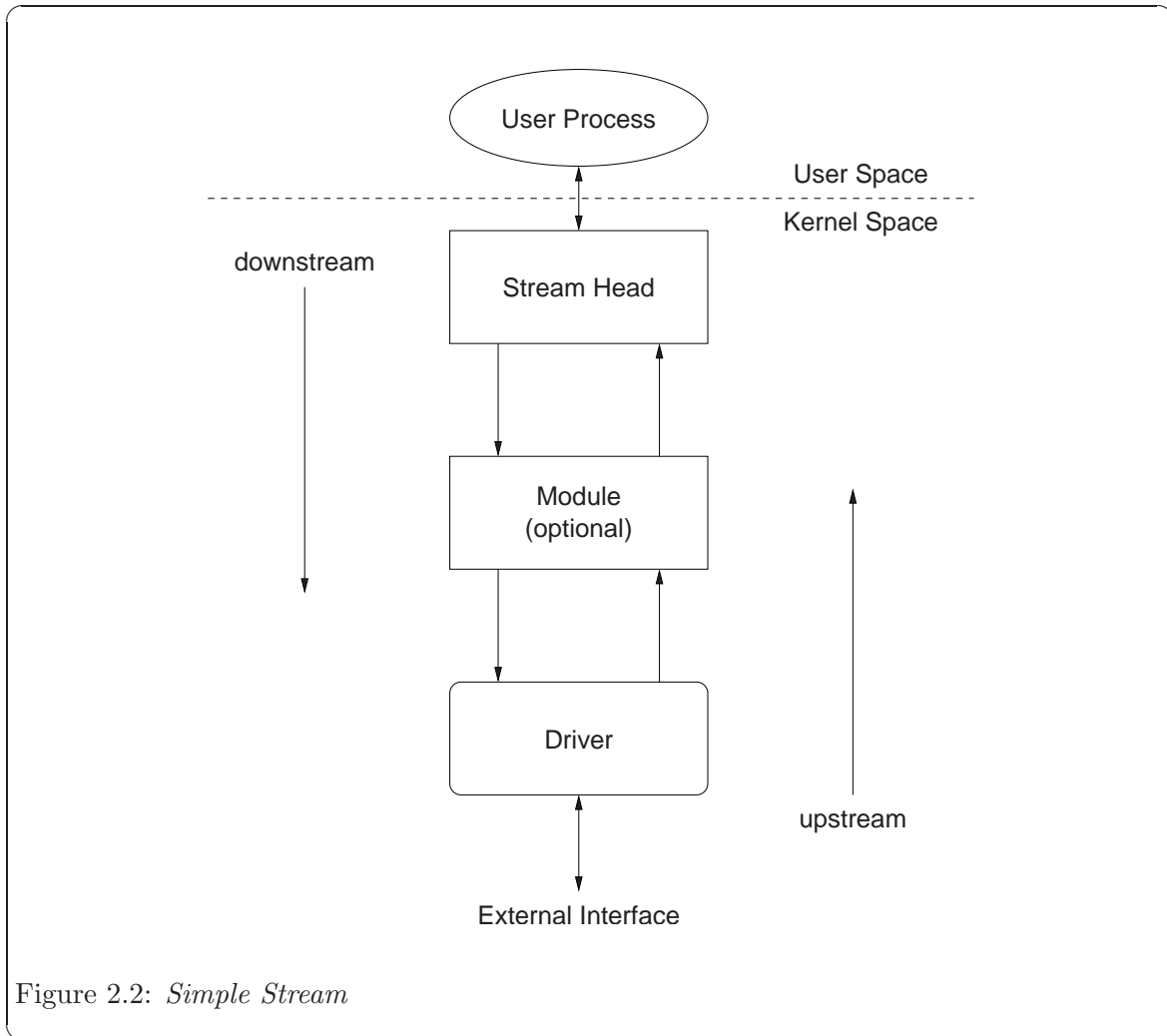
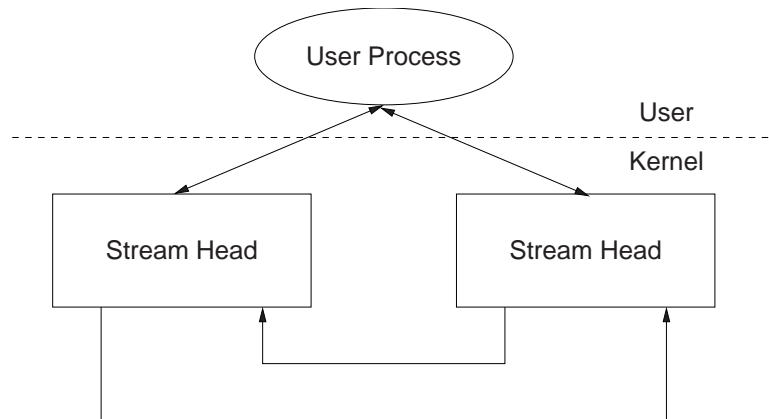


Figure 2.2: *Simple Stream*

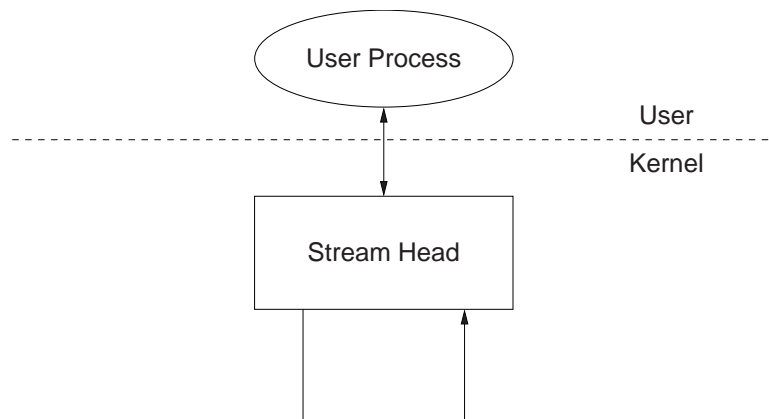
In the simple stream, the *downstream* flow is the flow from the User Process to the Driver that passes through any optional intervening Modules; the *upstream* flow is the flow from the Driver to the User Process, again through any intervening Modules.

It is possible to have a stream that does not contain a module or driver. There are two types of streams that do not contain a module or driver: a pipe and a *FIFO* (named pipe).

A *STREAMS*-based pipe is illustrated in [Figure 2.3](#).

Figure 2.3: *STREAMS-based Pipe*

A *STREAMS*-based *FIFO* is illustrated in [Figure 2.4](#).

Figure 2.4: *STREAMS-based FIFO (named pipe)*

Each *STREAMS* driver or module has a number of constituent pieces. Each *STREAMS* driver or module contains a stream table, stream administration, module information, module statistics and queue initialization information.

Each instance of a *STREAMS* driver or module has associated with it a stream head and a stream end (the driver). Each instance of a *STREAMS* driver or module contains a queue pair with a write (downstream) and read (upstream) queue. Each queue possibly has queue band information associated with the queue.

Each *STREAMS* queue and queue band can contain *STREAMS* messages. *STREAMS* messages are composed of message blocks, data blocks and data buffers in a chain-buffer arrangement.

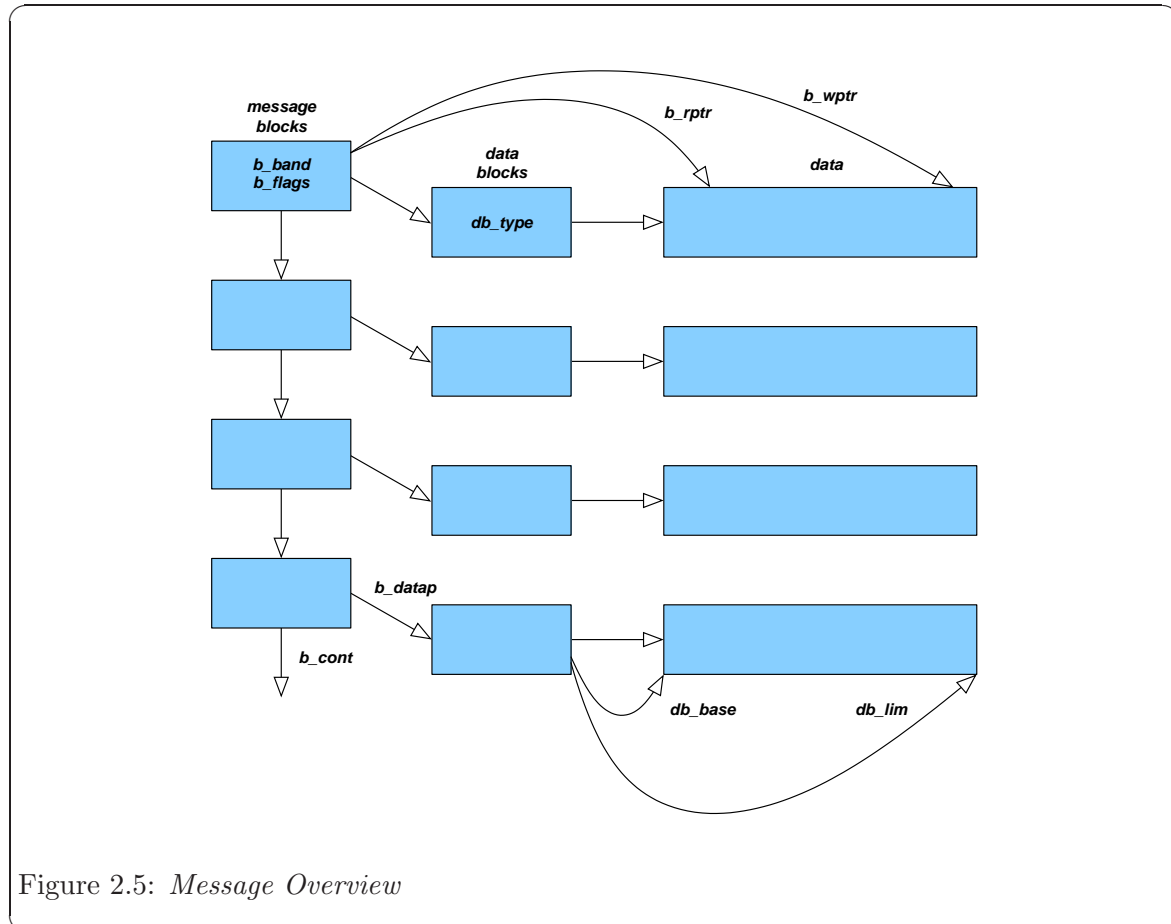


Figure 2.5 illustrates a *STREAMS* message. Messages are strings of message blocks. Each message block refers to a data block. Each data block has a data area associated with it. The data block has a message type ('*db_type*') associated with it. Message types can be one of the following:

Normal Messages

M_DATA	D	U	data to or from the user
M_PROTO	D	U	protocol primitive
M_BREAK	D	-	request a driver to send a break in the medium
M_CTL	D	U	used for inter-module and driver communication
M_DELAY	D	-	requests a real-time delay in output processing
M_IOCTL	D	-	passes a streams ioctl call
M_PASSFP	D	U	user be stream heads to pass file pointers to each other (e.g. pipes)

M_RSE	D	U	reserved
M_SETOPTS	-	U	set options on the stream head
M_SIG	-	U	sends a signal to the user

Priority Messages

M_COPYIN	-	U	copy data from user for transparent ioctl
M_COPYOUT	-	U	copy data to user for transparent ioctl
M_ERROR	-	U	sends a fatal error to the stream head
M_FLUSH	D	U	requests that driver and modules discard messages
M_HANGUP	-	U	indicates to the stream head that no more data can be read or written
M_IOCACK	-	U	acknowledges an ioctl request
M_IOCNAK	-	U	negatively acknowledges an ioctl request
M_IOCDATA	D	-	provides data requested by a M_COPYIN
M_PCPROTO	D	U	priority protocol primitive
M_PCRSE	D	U	reserved
M_PCSIG	-	U	sends a priority signal to the user
M_READ	D	-	sent by the stream head when the user issues an unsatisfied read(2) call
M_STOP	D	-	requests that a device stop its output
M_START	D	-	request that a device start its output
M_STARTI	D	-	requests that a device start its input
M_STOPI	D	-	requests that a device stop its input

Message functions consist of the following:

adjmsg(9)	trims bytes from the front or back of a message
allocb(9)	allocates a combined message and data block
bufcall(9)	calls a specified function when buffers become available
copyb(9)	copies a message block including the data
copymsg(9)	copies all the message blocks and data making up a message
datamsg(9)	tests if the message is a data message (M_DATA, M_PROTO, ...)
dupb(9)	duplicates a message block but references the same data
dupmsg(9)	duplicates all message blocks making up a message but references the same data in each message block
esballoc(9)	allocates a message block and data block but references a user-provided data and free function
freeb(9)	frees a message block
freemsg(9)	frees an entire message
linkb(9)	links a message block onto the end of a message
msgdsiz(9)	counts the bytes in M_DATA blocks in a message
pullupmsg(9)	places data from an entire message into a single message block
unbufcall(9)	cancels an earlier buffer callback request
unlinkb(9)	remove the first message block in a message

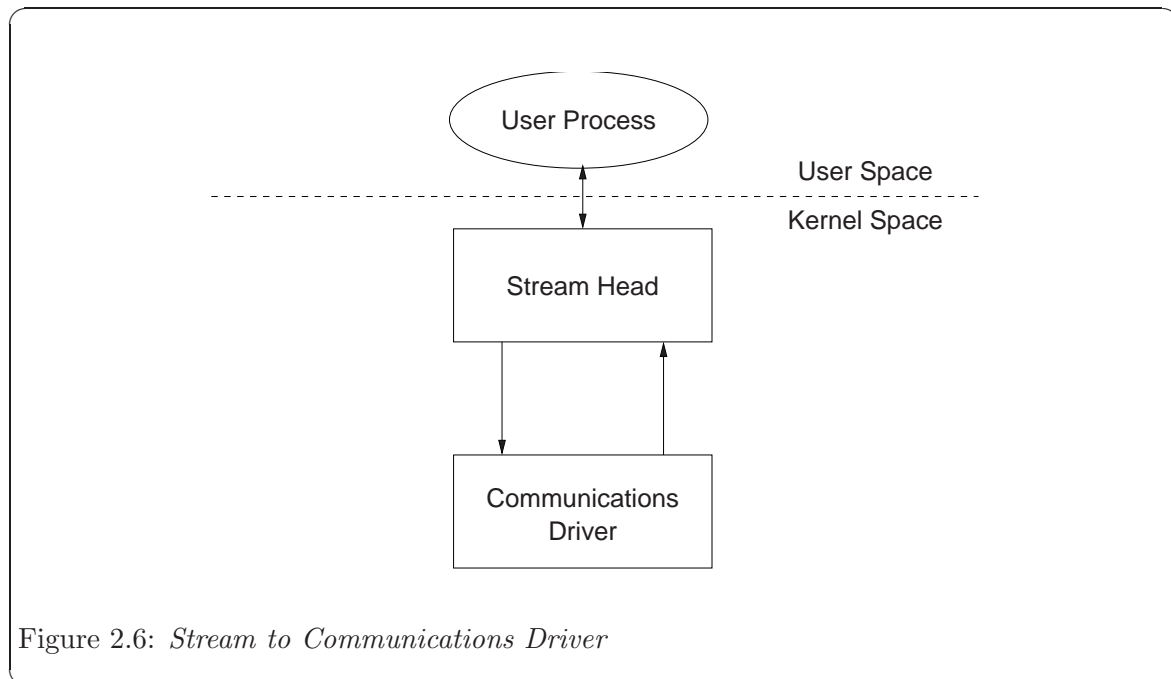
Messages can be priority or normal messages. Priority messages are delivered ahead of normal messages. Normal messages can be in one of 256 bands (0-255). Band 'n' messages

are delivered ahead of band ‘m’ messages where ‘n>m’. Queues are scheduled whenever messages are placed on the tail of the queue. Queues are scheduled whenever priority messages exist in a queue, regardless of whether they were placed on the tail or the head of the queue. Care must be taken not to return a priority message to the head of the queue in a service procedure or the service procedure will simply be rescheduled.

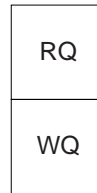
Buffers can be allocated as BPRI_HI, BPRI_MED, BPRI_LO and BRPI_WAITOK. In many *STREAMS* implementations, these priorities have no meaning.

A buffer that is smaller than a given size (FASTBUF) will be allocated more quickly. This is because the buffer is allocated coincident with the message block and data block.

When the system runs out of buffer, it will return failure to `allocb(9)` and `esballoc(9)`. `bufcall(9)` or `esbcall(9)` can be used to have the *STREAMS* subsystem call a client function when a buffer of the given size is available. Care must be taken with the use of `bufcall(9)` that deadlock situations do not arise.



Each module in a stream consists of a pair of queues: one for read (upstream) and one for write (downstream). Each queue is one of a pair of queues, see Figure 2.7. Where ‘q’ is one of the queues in the pair, the paired queue is identified by ‘OTHERQ(q)’. Where ‘q’ is the write queue in the pair, the read queue is identified by ‘RD(q)’. The read queue is for messages ultimately read by the user. Where ‘q’ is the read queue in the pair, the write queue is identified by ‘WR(q)’. The write queue is for messages first written by the user.

Figure 2.7: *Queue Pair Allocation*

Queue functions consist of the following:

<code>backq(9)</code>	obtains a pointer to the previous queue in a stream
<code>bcanput(9)</code>	tests for flow-control in a particular priority band
<code>canput(9)</code>	test flow-control
<code>enableok(9)</code>	marks a queue as being ready to be enabled
<code>flushband(9)</code>	removes all queued messages in a specified priority band
<code>flushq(9)</code>	removes all messages from a queue
<code>getadmin(9)</code>	finds the pointer to the <code>qi_qadmin()</code> function for a module
<code>getmid(9)</code>	looks up the module identifier
<code>getq(9)</code>	gets a message from a queue
<code>insq(9)</code>	inserts a message at a given point in a queue
<code>noenable(9)</code>	stops a queue from being scheduled
<code>OTHERQ(9)</code>	finds the sibling of a given queue
<code>putbq(9)</code>	puts back a message on a queue
<code>putctl(9)</code>	puts a control (e.g., <code>M_CTL</code>) message on a queue
<code>putctl1(9)</code>	puts a 1-byte control message on a queue
<code>putnext(9)</code>	passes a message to the next queue in a stream
<code>putq(9)</code>	puts a message on a queue
<code>qenable(9)</code>	schedule a queue
<code>qreply(9)</code>	passes a reply along a stream in the opposite direction
<code>qsize(9)</code>	returns the number of message on a queue
<code>RD(9)</code>	finds the read queue
<code>rmvq(9)</code>	removes a message from the middle of a queue
<code>strqget(9)</code>	queries the information from a queue
<code>WR(9)</code>	finds the write queue

The queue open procedure is called whenever a driver special character device is opened. It is also called when a module is pushed onto a stream.

The queue close procedure is called when a driver special character device is closed for the last time. It is also called when a module is popped from a stream.

The queue put procedure is called whenever a previous queue in the stream passes a message to the queue, or a message is otherwise placed on the queue with the `put(9)` function.

The queue service procedure is called when there are messages on the queue to be serviced and the *STREAMS* subsystem has scheduled the queue for service.

Put and service procedures must handle `M_DATA`, `M_PROTO`, `M_PCPROTO`, `M_FLUSH` and `M_IOCTL` messages. Other messages are more or less optional.

Module put and service procedures should pass any unrecognized message types along. Module put and service procedures should pass any unrecognized `M_IOCTL ioctl(2)` commands along. Modules must perform canonical flushing in response to `M_FLUSH` messages. This includes passing `M_FLUSH` messages along downstream or upstream.

Driver put and service procedures should discard any unrecognized message types. Driver put and service procedures should negatively acknowledge (`M_IOCNAK`) any unrecognized `M_IOCTL` commands. Drivers must perform canonical flushing in response to `M_FLUSH` messages. This includes sending `M_FLUSH` message that arrive on their way downstream to the upstream queues (`qreply(9)`).

Queue scheduling pertains to when a queue's service procedure is schedule to run by the kernel. When a queue's service procedure is scheduled to run, the queue is considered enabled. When a queue's service routine is prohibited from running, it is considered disabled. Only queues that have service procedures can become enabled. Queues with only `put(9)` procedures are never enabled.

Queues are enabled whenever a priority messages is placed on the queue. Queues are enable whenever `qenable(9)` is called explicitly. Queues are also enabled as a result of back-enabling. A queue is back-enabled when it previously tested the next queue for flow control and found the queue flow controlled (the queue had passed its high water mark); the queue which was previously test is now no longer flow controlled (it has fallen below its low water mark); the queue which previously tested can now put to the queue which was flow controlled.

Queues are disabled whenever the queue service procedure has run. Queues are disabled whenever `noenable(9)` has been called explicitly. Queues are disabled whenever they are empty.

Flow control pertains to when a queue will accept messages put to it from above and below. When the number of bytes in a queue passes the high water mark associated with the queue, the queue is considered flow controlled. Flow controlled queues will fail a `canput(9)` call. When the number of bytes in a flow controlled queue falls below the low water mark, the queue is considered available to process messages again. Only queues that have a service procedure can become flow controlled. Only queues that have service procedures need check for flow control downstream.

It is the responsibility of a module performing a `putq(9)`, a `putnext(9)` or a `qreply(9)` to a queue to check for flow control with `canput(9)`, `canputnext(9)`, `bcanput(9)` or `bcanputnext(9)` before placing the message on a queue. Modules that receive messages when they are flow controlled might discard them. Flow control kicks in when the number of byte queued passes the queue's high water mark.

Back-enabling occurs when a queue that was previously flow controlled, drops below its low water mark, and was tested while it was flow controlled. Flow control can be difficult to accomplish correctly in drivers and across multiplexers.

STREAMS drivers, modules and multiplexing drivers are described with a series of statically allocated structures beginning with the driver switch table or module switch table entries.

2.2.1 Stream Administration

Some *STREAMS* implementation also provide some sort of *STREAMS* administration structure that contains more specific information about a *STREAMS* driver or module, such as the name of the driver or module, synchronization, administrative flags, version, etc. These administrative structures are normally passed to the registration procedure to register a driver or module. Implementations that function this way are *AIX*, *OSF/1*, *HP-UX* and *UnixWare*.

AIX uses the `strconf_t` structure and calls the `str_install_AIX(9)` registration function. *OSF/1* uses the `streamadm(9)` structure and calls the `strmod_add(9)` and `strmod_del(9)` registration functions. *HP-UX* uses the `stream_inst(9)` structure and calls the `str_install_HPUX(9)` and `str_uninstall(9)` registration functions.

The ‘`sys/strconf.h`’ header file contains definitions for both the *STREAMS* administrative structures as well as declarations and function prototypes for the registration functions.

Other *STREAMS* implementations use the character device switch table or module switch table entries themselves. The partially completed entry might still be passed to the registration function like a *STREAMS* administrative structure. Implementations that function this way are *Solaris* and *Linux Fast-STREAMS*.

Solaris uses a plethora of statically allocated structures to describe *STREAMS* drivers and modules. Several of these structures correspond to the character device (well, character and block device) switch table entry as well as the module switch table entry. *Linux Fast-STREAMS* uses the actual character device switch table and module switch table structures and calls the `register_strdrv(9)` and `register_strmod(9)` registration functions.

Some implementations use a registration function alone and do not pass a structure. Implementations that function this way are *LiS*.

LiS simply calls the `lis_register_strdrv(9)` and `lis_register_strmod(9)` registration functions.

2.2.2 Driver Switch Table

Each *STREAMS* device or pseudo-device driver or multiplexing driver registers itself with the *STREAMS* subsystem. Nevertheless, to allow user processes to open the *STREAMS* device requires that the device be entered into the system device switch tables. *SVR 3* had a device switch table and many other *UNIX* implementations do as well. **Linux** does not use a device switch table. Rather it uses character device lists hashed on device number, or a file system mechanism such as the device file system (‘`devfs`’) or ‘`udev`’. *Linux Fast-STREAMS* registers devices with the necessary **Linux** mechanism and then implements an internal device switch table as a list hashed on device number.

Normally, *STREAMS* implementations have some mechanism for registering a driver with the character device switch table or other structures used for opening *STREAMS* devices. Some configuration procedure take structures (e.g. *Solaris*), some do not (e.g. *LiS*). Each configuration technique is unique to that implementation. *Linux Fast-STREAMS* supports its own registration technique as well as several of the other more common implementation methods. *Linux Fast-STREAMS* has its own registration procedure that consists of first

completing a static `cdevsw(9)` structure for entry into the `cdevsw_list(9)`, which is *Linux Fast-STREAMS*'s equivalent of a character device switch table.

The character device switch table entry structure (`cdevsw(9)`) structure takes the following information:

<code>d_list</code>	list of all <code>cdevsw</code> structures
<code>d_hash</code>	list of module hashes in slot
<code>d_name</code>	driver name
<code>d_str</code>	pointer to <code>streamtab(9)</code> for driver
<code>d_flag</code>	driver flags
<code>d_modid</code>	driver module identifier
<code>d_count</code>	open count
<code>d_sqlvl</code>	synchronization level
<code>d_syncq</code>	synchronization queue
<code>d_kmod</code>	kernel module
<code>d_major</code>	base major device number
<code>d_inode</code>	specfs inode
<code>d_mode</code>	inode mode
<code>d_fop</code>	file operations
<code>d_majors</code>	major device nodes for this device
<code>d_minors</code>	minor device nodes for this device
<code>d_apush</code>	autopush list
<code>d_plinks</code>	permanent links for this driver
<code>d_stlist</code>	stream head list for this driver

Only the driver name, `streamtab(9)` pointer, flags and synchronization level need be completed by the module writer: the remaining fields in the `cdevsw(9)` structure will be populated by the registration procedure. The procedure used by *Linux Fast-STREAMS* for registering *STREAMS* drivers is the `register_strdrv(9)` kernel function.

2.2.3 Module Switch Table

Each *STREAMS* module registers itself with the *STREAMS* subsystem and is assigned a unique module identification number. Even *STREAMS* device or pseudo-device driver's are assigned a unique module identification number. In most *STREAMS* implementations, modules are entered by module identification number into a module switch table. *Linux Fast-STREAMS* implements an internal module switch table as a list hashed on module identifier.

Normally, *STREAMS* implementations have some mechanism for registering a module with the module switch table or other structures used for opening *STREAMS* modules. Some configuration procedures take structures (e.g. *Solaris*), some do not (e.g. *LiS*). Each configuration technique is unique to that implementation. *Linux Fast-STREAMS* supports its own configuration technique as well as several of the other more common implementation methods. *Linux Fast-STREAMS* has its own registration procedure that consists of first completing a static `fmodsw(9)` structure for entry into the `fmodsw_list(9)`, which is *Linux Fast-STREAMS*'s equivalent of a module switch table.

The module switch table entry structure (`fmodsw(9)`) structure takes the following information:

<code>f_list</code>	list of all <code>fmodsw</code> structures
<code>f_hash</code>	list of module hashes in slot
<code>f_name</code>	module name
<code>f_str</code>	pointer to <code>streamtab(9)</code> for module
<code>f_flag</code>	module flags
<code>f_modid</code>	module identifier
<code>f_count</code>	open count
<code>f_sqlvl</code>	synchronization level
<code>f_syncq</code>	synchronization queue
<code>f_kmod</code>	kernel module

Only the module name, `streamtab(9)` pointer, flags and synchronization level need be completed by the module writer: the remaining fields in the `fmodsw(9)` structure will be populated by the registration procedure. The procedure used by *Linux Fast-STREAMS* for registering *STREAMS* modules is the `register_strmod(9)` kernel function. It is no coincidence that the `fmodsw` structure is identical to the first portion of the `cdevsw` structure.

2.2.4 Stream Table

Each *STREAMS* driver or module has an external entry point into a streams table. The stream table structure is the jumping off point for all driver or module specific data structures that describe the *STREAMS* driver or module. Each *STREAMS* driver or module that is unique within the *STREAMS* subsystem has a unique stream table, regardless of which device major numbers, module identifiers, or other external registrations have been performed on behalf of the driver or module.

The stream table structure (`streamtab(9)`) contains the following information:

<code>st_rdinit</code>	read queue init structure pointer
<code>st_wrinit</code>	write queue init structure pointer
<code>st_muxrinit</code>	multiplexer lower read queue init structure pointer
<code>st_muxwinit</code>	multiplexer lower write queue init structure pointer

Only the `st_rdinit` and `st_wrinit` members need be completed for a normal module or driver. For a multiplexing driver that accepts the `I_LINK` or `I_PLINK` commands, the lower queue initialization information, `st_muxrinit` and `st_muxwinit` must be completed as well.

2.2.5 Queue Initialization

Each stream table (driver or module) has associated with it queue initialization information for the upstream and downstream queues. If the driver is a multiplexing driver, it may also have associated with the stream table upstream and downstream queue initialization information for the lower queue pair.

The queue initialization structure provides function pointers to the put, service, open, close and administrative procedures for the queue to which the initialization structure belongs. Queue initialization structures are bound to a specific upper or lower, upstream or downstream queue using the stream table. Up to four queue initialization structures can exist

for a driver or module, or, it is possible that all four queues could share a single queue initialization structures.¹

The queue initialization structure (`qiinit(9)`) contains the following information:

<code>qi_putp</code>	put procedure
<code>qi_srvp</code>	service procedure
<code>qi_qopen</code>	each open (read queue only)
<code>qi_qclose</code>	last close (read queue only)
<code>qi_qadmin</code>	administrative procedure (not used)
<code>qi_minfo</code>	module information
<code>qi_mstat</code>	module statistics

2.2.6 Module Information

Each queue initialization structure points to a module information structure that provides some queueing parameters for the stream. Nevertheless, only one module information structure is necessary for the stream. The module information structure contains the module identifier, the module name, the minimum and maximum packet sizes accepted on the queue from the stream head, and the high and low water marks for flow control. If any of this later information needs to differ on a queue basis (upper or lower multiplex queues considered as well), then the module information structure must be separate for each queue.

The module information structure (`module_info(9)`) contains the following information:

<code>mi_idnum</code>	module identification number
<code>mi_idname</code>	module name
<code>mi_minpsz</code>	minimum packet size accepted
<code>mi_maxpsz</code>	maximum packet size accepted
<code>mi_hiwat</code>	high water mark
<code>mi_lowat</code>	low water mark

2.2.7 Module Statistics

Each queue initialization structure points to an optional module statistics structure that collects statistics for the stream.

The module statistics structure (`module_stat(9)`) contains the following information:

<code>ms_pcmt</code>	calls to <code>qi_putp</code>
<code>ms_scmt</code>	calls to <code>qi_srvp</code>
<code>ms_ocmt</code>	calls to <code>qi_qopen</code>
<code>ms_ccmt</code>	calls to <code>qi_qclose</code>
<code>ms_acmt</code>	calls to <code>qi_qadmin</code>
<code>ms_xptr</code>	module private statistics pointer
<code>ms_xsize</code>	size of module private statistics area
<code>ms_flags</code>	boolean statistics, for future use

¹ Only the open and close procedures indicated in the queue initialization structure associated with an upper upstream (read) queue will be used to open and close an instance of the driver or module. Open and close function pointers in the other queue initialization structures will be ignored.

2.2.8 Stream Head

A stream head is a dynamic structure that is created whenever a *STREAMS* driver (stream end) is instantiated.

The stream head accepts the following input-output controls:

I_PUSH	push a module
I_POP	pop a module
I_SETSIG	receive a {SIGPOLL} signal when an event occurs
I_FDINSERT	pass informationa about a stream
I_STR	generate an ioctl(2) to a streams module
I_SENDFD	send a file descriptor down a stream
I_LINK	link a stream under a multiplexer
I_PLINK	permanently link a stream under a multiplexer
I_LOOK	get name of module below stream head
I_FLUSH	flush a stream
I_FLUSHBAND	flush a stream in a given band
I_GETSIG	return events that generate {SIGPOLL}
I_FIND	find a particular module in a stream
I_PEEK	read first message on a stream
I_SRDOPT	set read options
I_GRDOPT	get read options
I_SWROPT	set write options
I_GWROPT	get write options
I_RECVFD	retrieve a passed file descriptor
I_LIST	list all modules in a stream
I_ATMARK	see if stream head message has been marked
I_CKBAND	check if message of a particular band is writable
I_GETBAND	get priority band of first message at stream head
I_CANPUT	cehck if given band is writable
I_SETCLTIME	set the time that <i>STREAMS</i> will wait while closing
I_GETCLTIME	get the time that <i>STREAMS</i> will wait while closing
I_UNLINK	unlink a stream from beneath a multiplexer
I_PUNLINK	unlink a permanent stream from beneath a multiplexer
SO_ALL	set all old options
SO_READOPT	set read option
SO_WROFF	set write offset
SO_MINPSZ	set minimum packet size
SO_MAXPSZ	set maximum packet size
SO_HIWAT	set high water mark
SO_LOWAT	set low water mark
SO_MREADON	set read notification on
SO_MREADOFF	set read notification off
SO_NDELO	old <i>TTY</i> semantics for NDELAY reads and writes
SO_NDELOFF	<i>STREAMS</i> semantics for NDELAY reads and writes
SO_ISTTY	the stream is acting as a terminal

SO_ISNTTY	the stream is not acting as a terminal
SO_TOSTOP	stop on background writes to this stream
SO_TONSTOP	do not stop on background writes to this stream
SO_BAND	water marks affect band
SO_DELIM	messages are delimited
SO_NODELIM	turn off delimiters
SO_STRHOLD	<i>UnixWare</i> and <i>Solaris</i> only: enable strwrite message coalescing
SO_ERROPT	<i>Solaris</i> only
SO_LOOP	<i>UnixWare</i> only
SO_COPYOPT	<i>Solaris</i> only: user io copy options
SO_MAXBLK	<i>Solaris</i> only: maximum block size
IOCWAIT	ioctl in progress
RSLEEP	process sleeping on read
WSLEEP	process sleeping on write
STRPRI	priority message waiting
STRHUP	stream is hung up
STWOPEN	stream head open in progress
STPLEX	stream linked under mux
STRISTTY	stream is a terminal
STRDERR	M_ERROR for read received
STRWRERR	M_ERROR for write received
STRCLOSE	wait for strclose to complete
SNDMREAD	send M_READ msg when read issued
STRHOLD	coalesce written messages
STRMSIG	M_SIG at head of queue
STRDELIM	generate delimited messages
STRTOSTOP	stop timeout
STRISFIFO	stream is a fifo
STRISPIPE	stream is a <i>STREAMS</i> pipe
STRISOCK	stream is a <i>STREAMS</i> socket
STFROZEN	stream is frozen

2.2.9 Queue

A pair of queues are allocated, initialized, assigned and linked into a stream each time that a *STREAMS* driver or module is instantiated. A pointer to the upstream (read) queue in the queue pair is the common method for passing a handle to a driver or module instance.

2.2.10 Queue Band

Queue bands are allocated and initialized in association with queues on demand. Many *STREAMS* drivers or modules never have queue bands allocated because they do not pass priority banded messages, only normal messages.

2.2.11 Message Block

A message block is a structure that references a data block and data buffer. Each message block is a view into a data buffer and has a start (read pointer) and finish (write pointer) within the buffer. Message block can be chained together to form larger segmented messages.

2.2.12 Data Block

Data blocks are a description of a data buffer external to the buffer itself. Each data block describes the limits of the data buffer, its base, and size. Each data block also hold a number of flags and contains a data block type describing the type of data contained in the data buffer. Data blocks also contain any necessary information concerning the functions and arguments necessary to free the data buffer. Also, data blocks maintain reference counts of the number of message blocks that reference the data buffer.

2.2.12.1 Data Block Types

2.2.13 Data Buffer

Data buffers can be allocated within the *STREAMS* buffer pools using the `alloca(9)` function or can be allocated externally and associated with a data and message block using the `esballoc(9)` function.

2.2.14 User Credentials

2.3 Application Interface

2.3.1 System Calls

<code>close(2)</code>	—
<code>fattach(2)</code>	—
<code>fcntl(2)</code>	—
<code>fdetach(2)</code>	—
<code>getmsg(2)</code>	—
<code>getpmsg(2)</code>	—
<code>ioctl(2)</code>	—
<code>isastream(2)</code>	—
<code>open(2)</code>	—
<code>pipe(2)</code>	—
<code>poll(2)</code>	—
<code>putmsg(2)</code>	—
<code>putpmsg(2)</code>	—
<code>read(2)</code>	—
<code>readv(2)</code>	—
<code>select(2)</code>	—
<code>write(2)</code>	—
<code>writew(2)</code>	—

2.4 Kernel Level Facilities

2.4.1 Stream Head

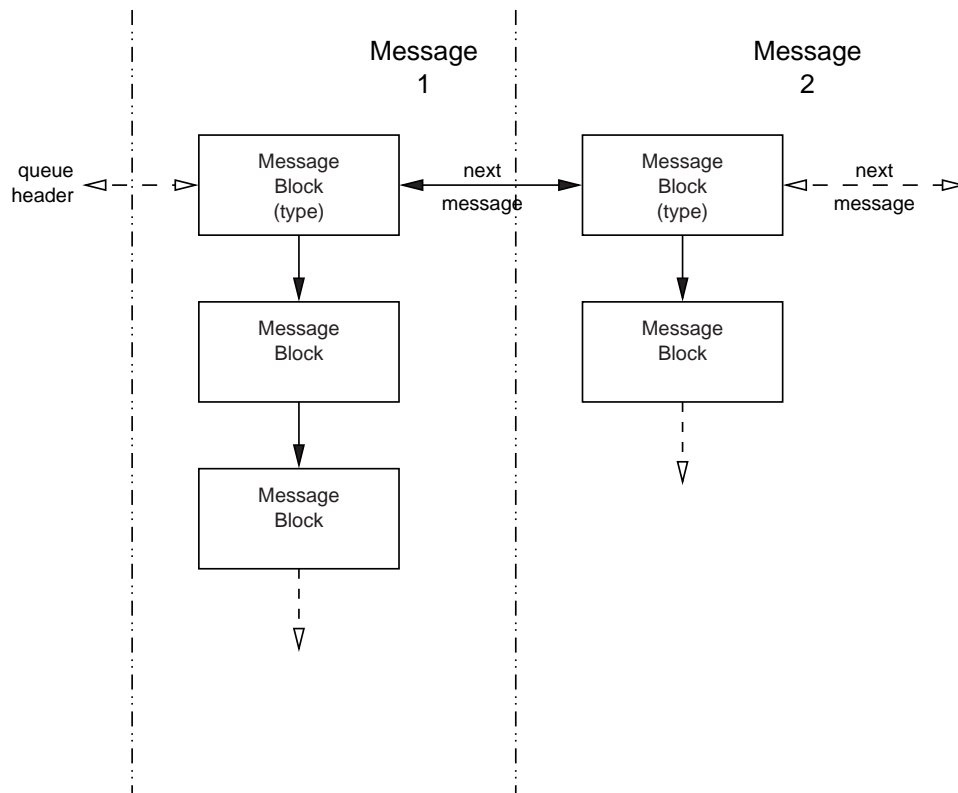
2.4.2 Modules

2.4.3 Drivers

2.4.4 Messages



Figure 2.8: *A Message*

Figure 2.9: *Messages on a Message Queue*

2.4.4.1 Message Types

Normal Messages

M_DATA	D	U	Normal data.
M_PROTO	D	U	Normal protocol information.
M_BREAK	D	–	Break request.
M_PASSFP	D	U	Passed file pointer.
M_EVENT	–	U	Normal event. (<i>Solaris</i> only)
M_SIG	–	U	Signal.
M_DELAY	D	–	Delay request.
M_CTL	D	U	Normal inter-module control message.
M_IOCTL	D	–	Input-Output control.
M_SETOPTS	–	U	Normal set queue options.
M_RSE	D	U	Normal reserved.
M_TRAIL	–	–	(<i>UnixWare</i> only)
M_BACKWASH	D	–	(<i>AIX</i> only)

Priority Messages

M_IOCACK	–	U	Input-Output control acknowledgement (result).
M_IOCNAK	–	U	Input-Output control negative acknowledgement (error).
M_PCPROTO	D	U	Priority protocol information.
M_PCSIG	–	U	Priority signal.
M_READ	D	–	Read request.
M_FLUSH	D	U	Flush queue request.
M_STOP	D	–	Stop output request.
M_START	D	–	Start output request.
M_HANGUP	–	U	Hangup.
M_ERROR	–	U	Fatal error.
M_COPYIN	–	U	Input-Output control copyin request.
M_COPYOUT	–	U	Input-Output control copyout request.
M_IOCDATA	D	–	Input-Output control copied in data.
M_PCRSE	D	U	Priority reserved.
M_STOPI	D	–	Stop input request.
M_STARTI	D	–	Start input request.
M_PCCTL	D	U	Priority inter-module control message. (<i>UnixWare</i> only)
M_PCSETOPTS	–	U	Priority set queue options. (<i>UnixWare</i> only)
M_PCEVENT	–	U	Priority event. (<i>Solaris</i> only)
M_UNHANGUP	–	U	Hangup corrected. (<i>Solaris</i> and <i>OSF/1</i> only)
M_NOTIFY	–	U	(<i>OSF/1</i> and <i>HP-UX</i> only)
M_HPDATA	D	U	(<i>HP-UX</i> and <i>MacOT</i> only)
M_LETSPLAY	–	U	(<i>AIX</i> only)
M_DONTPLAY	D	–	(<i>AIX</i> only)
M_BACKDONE	D	–	(<i>AIX</i> only)

2.4.5 Message Queueing Priority

2.4.6 Queues

2.4.7 Multiplexing

2.4.8 Multithreading

2.5 Subsystems

2.5.1 Logging

The kernel logger and system logger on a **Linux** system is the native logging implementation. These are *BSD*-style loggers. *Linux Fast-STREAMS* provides a *STREAMS*-based logger as well. The advantage of a *STREAMS*-based logger is for trace and error logging *STREAMS* drivers and modules using the `strlog` facility. This facility includes a device driver (the *STREAMS* log device, `log(4)`), a kernel function (`strlog(9)`), and two user space daemon processes (`strace(8)` and `strerr(8)`). The *STREAMS* logger has the unique capability

that it can filter trace messages and only incur the cost of tracing and logging those messages that are filtered out (at the kernel level). This allows a large degree of trace logging to be available, yet, only a small proportion of the available trace messages generated to logs.

The *STREAMS* logger `log(4)`, `strlog(9)` command, and `strace(8)` and `strerr(8)` loggers are all available in the base `streams-0.7a.3` package.²

2.5.2 Administrative Driver

Linux Fast-STREAMS provides a *STREAMS* administration facility. This facility consists of a driver (the *STREAMS Administrative Driver*, `sad(4)`), several autopush kernel functions (`autopush_add(9)`, `autopush_del(9)`, `autopush_find(9)`, `autopush_vml(9)`), and the `autopush(8)`, `insf(8)`, `strload(8)`, `strsetup(8)`, `strinfo(8)`, `scls(8)` and other administrative commands.

2.5.3 Terminal I/O

By default, terminal I/O in a **Linux** system does not use *STREAMS*. It, in fact, uses a *STREAMS* ‘pty’ emulation. However, this is only an emulation and does not provide all *STREAMS* facilities. It is not possible, for example, on a **Linux** ‘pty’ to push or pop modules from a terminal. The **Linux** ‘pty’ only provides emulation for read/write and ioctl calls.

A separate auxiliary package for *Linux Fast-STREAMS* that provides true *STREAMS*-based UNIX’98 compliant pseudo terminals is available. This is the `strpty-0.7a.1` package also available from [The OpenSS7 Project](#).³

There are three user commands available in the *Linux Fast-STREAMS* package that are useful for terminal input-output: these are the `strchg(1)` `strconf(1)` and `strreset(1)` user commands.

2.5.4 Pipes

Linux Fast-STREAMS supports *STREAMS*-based pipes as an optional feature. *STREAMS*-based pipes operate as described in the *UNIX SVR4.2 Operating System API Reference* and the *UNIX System V Release 4 Programmer’s Guide: STREAMS*.

By default, pipes created on **Linux** by the `pipe(2)` system call are not *STREAMS*-based. To get *STREAMS*-based pipes, configure *Linux Fast-STREAMS* with configuration parameter `--enable-streams-fifos`.

Linux Fast-STREAMS also provides a character device based pipe facility using the `spx(4)` driver.⁴

² I have been considering removing these facilities into a common `strutil-0.7a.1` package that could work with *LiS* as well, perhaps.

³ Actually, this package is not yet available. I will work on it someday when it is necessary.

⁴ I am considering taking pipes and FIFOs out of the base `streams-0.7a.3` package and placing them in their own `strpipe-0.7a.1` package.

2.5.5 FIFOs

Linux Fast-STREAMS supports *STREAMS*-based FIFOs (named pipes) as an optional feature. *STREAMS*-based FIFOs operate as described in the *UNIX SVR4.2 Operating System API Reference* and the *UNIX System V Release 4 Programmer's Guide: STREAMS*. By default, FIFOs created on **Linux** with the `mknod(8)` utility are not *STREAMS*-based. To get *STREAMS*-based FIFOs, configure *Linux Fast-STREAMS* with configuration parameter `--enable-streams-fifos`.⁵

2.5.6 Networking

By default, networking in a **Linux** system does not use *STREAMS*. It uses the native Linux *BSD Sockets* approach. Some **GNU/Linux** distributions do provide an *iBCS* (*Intel Binary Compatibility Suite*) that provides *XTI/TLI* networking, however, this is only a *STREAMS* emulation and cannot push or pop protocol modules from a stream. Also, **Linux** *iBCS* only provides emulation for read/write and `ioctl` calls.

A separate auxiliary package for *Linux Fast-STREAMS* that provides a true *STREAMS*-based *UNIX'98* compliant *XTI/TLI* library is available. This is the `strxnet-0.9.2.5` package also available from [The OpenSS7 Project](#).

To be able to open *INET* streams also requires the `strinet-0.9.2.1` package also available from [The OpenSS7 Project](#). This package provides *STREAMS*-based networking by providing a specialized *STREAMS* driver that internally opens a **Linux** native *BSD Socket* and translates *STREAMS* messages to and from the internal *Socket*.

⁵ I am considering taking pipes and FIFOs out of the base `streams-0.7a.3` package and placing them in their own `strpipe-0.7a.1` package.

3 STREAMS Mechanism

3.1 STREAMS Mechanism Overview

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

In addition to the traditional `ioctl` commands and system calls, there are other system calls used by *STREAMS*. The `poll(2)` system call enables a user to poll multiple *Streams* for various events. The `putmsg(2)` and `getmsg(2)` system calls enable users to send and receive *STREAMS* messages, and are suitable for interacting with *STREAMS* modules and drivers through a service interface.

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

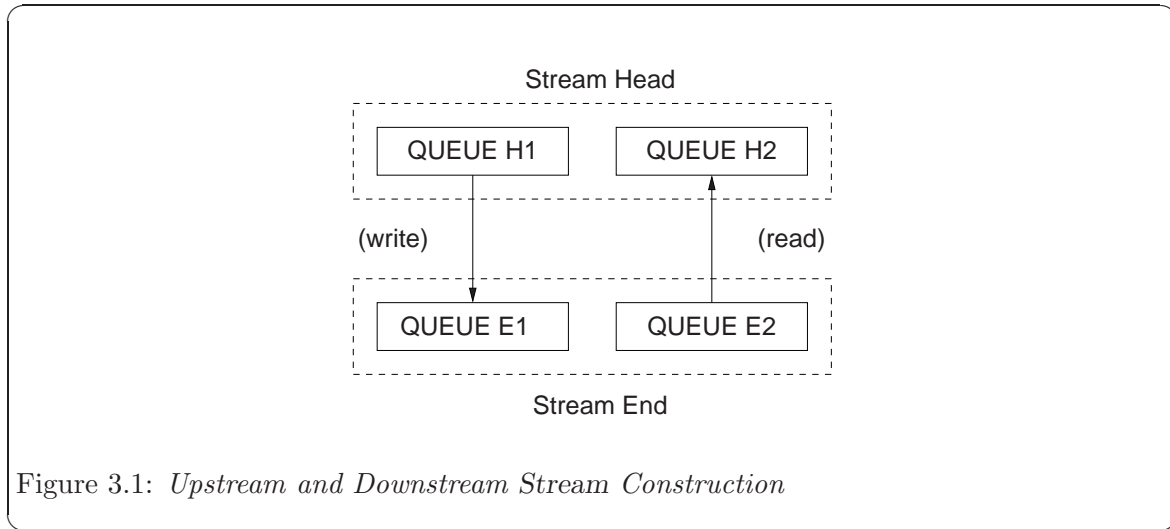
3.1.1 STREAMS System Calls

The *STREAMS*-related system calls are:

<code>open(2)</code>	Open a <i>Stream</i>
<code>close(2)</code>	Close a <i>Stream</i>
<code>read(2)</code>	Read data from a <i>Stream</i>
<code>write(2)</code>	Write data to a <i>Stream</i>
<code>ioctl(2)</code>	Control a <i>Stream</i>
<code>getmsg(2)</code>	Receive a message at the <i>Stream</i> head
<code>putmsg(2)</code>	Send a message downstream
<code>poll(2)</code>	Notify the application program when selected events occur on a <i>Stream</i>
<code>pipe(2)</code>	Create a channel that provides a communication path between multiple processes.

3.2 STREAMS Stream Construction

STREAMS constructs a *Stream* as a linked list of kernel resident data structures. The list is created as a set of linked queue pairs. The first queue pair is the head of the *Stream* and the second queue pair is the end of the *Stream*. The end of the *Stream* represents a device driver, pseudo device driver, or the other end of a *STREAMS*-based pipe. Kernel routines interface with the *Stream head* to perform operations on the *Stream*. [Figure 3.1](#) depicts the upstream (read) and downstream (write) portions of the *Stream*. Queue ‘H2’ is the upstream half of the *Stream head* and queue ‘H1’ is the downstream half of the *Stream head*. Queue ‘E2’ is the upstream half of the *Stream end* and queue ‘E1’ is the downstream half of the *Stream end*.



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

[Figure 3.2](#) the data structures forming each queue: `queue`, `qinit`, `qband`, `module_info`, and `module_stat`. The `qband` structures have information for each priority band in the queue. The queue data structure contains various modifiable values for that queue. The `qinit` structure contains a pointer to the processing procedures, the `module_info` structure contains initial limit values, and the `module_stat` structure is used for statistics gathering. Each queue in the queue pair contains a different set of these data structures. There is a `queue`, `qinit`, `module_info`, and `module_stat` data structure for the upstream portion of the queue pair and a set of data structures for the downstream portion of the pair. In some situations, a queue pair may share some or all of the data structures. For example, there may be a separate `qinit` structure for each queue in the pair and one `module_stat` structure that represents both queues in the pair. These data structures are described in [Appendix A \[STREAMS Data Structures\]](#), page 275.

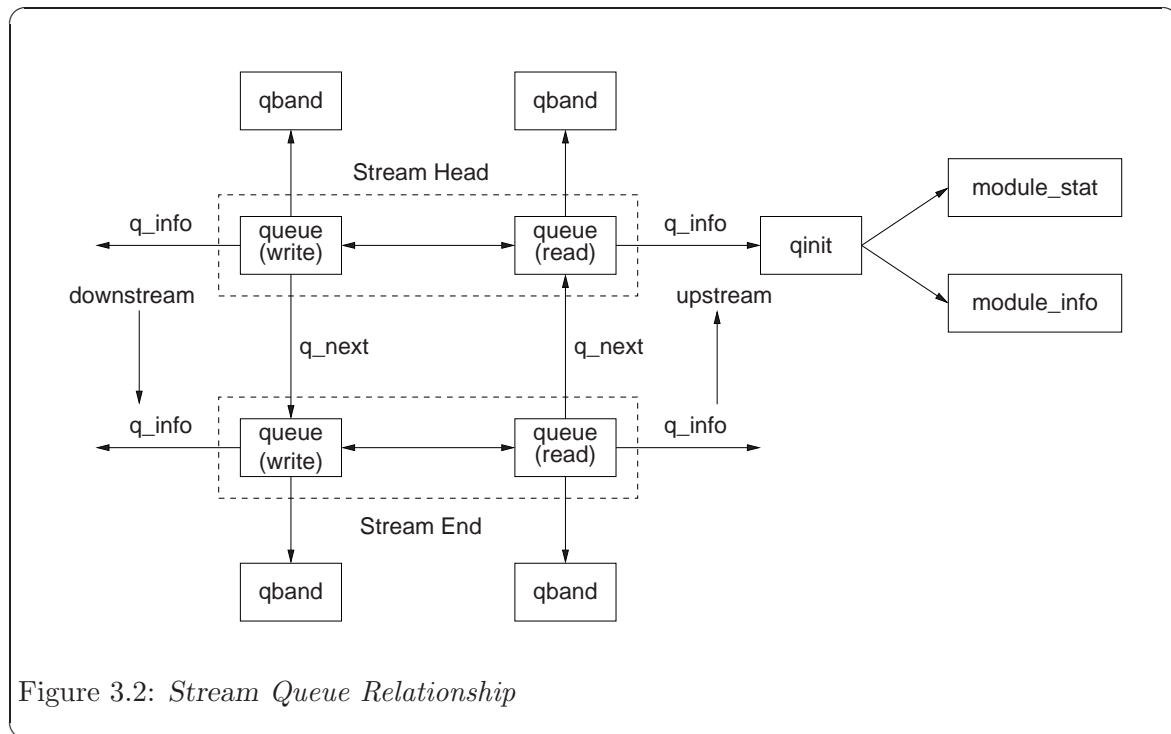


Figure 3.2 shows two neighboring queue pairs with links (solid vertical arrows) in both directions. When a module is pushed onto a *Stream*, *STREAMS* creates a queue pair and links each queue in the pair to its neighboring queue in the upstream and downstream direction. The linkage allows each queue to locate its next neighbor. This relation is implemented between adjacent queue pairs by the *q_next* pointer. Within a queue pair, each queue locates its mate (see dashed arrows in Figure 3.2) by use of *STREAMS* macros, since there is no pointer between the two queues. The existence of the *Stream head* and *Stream end* is known to the queue procedures only as destinations towards which messages are sent.

3.2.1 Opening a STREAMS Device File

One way to construct a *Stream* is to open [see `open(2)`] a *STREAMS*-based driver file (see [7]Figure:2.3 [8]Opened *STREAMS*-based Driver). All entry points into the driver are defined by the `streamtab(9)` structure for that driver. The `streamtab` structure has a format as follows:

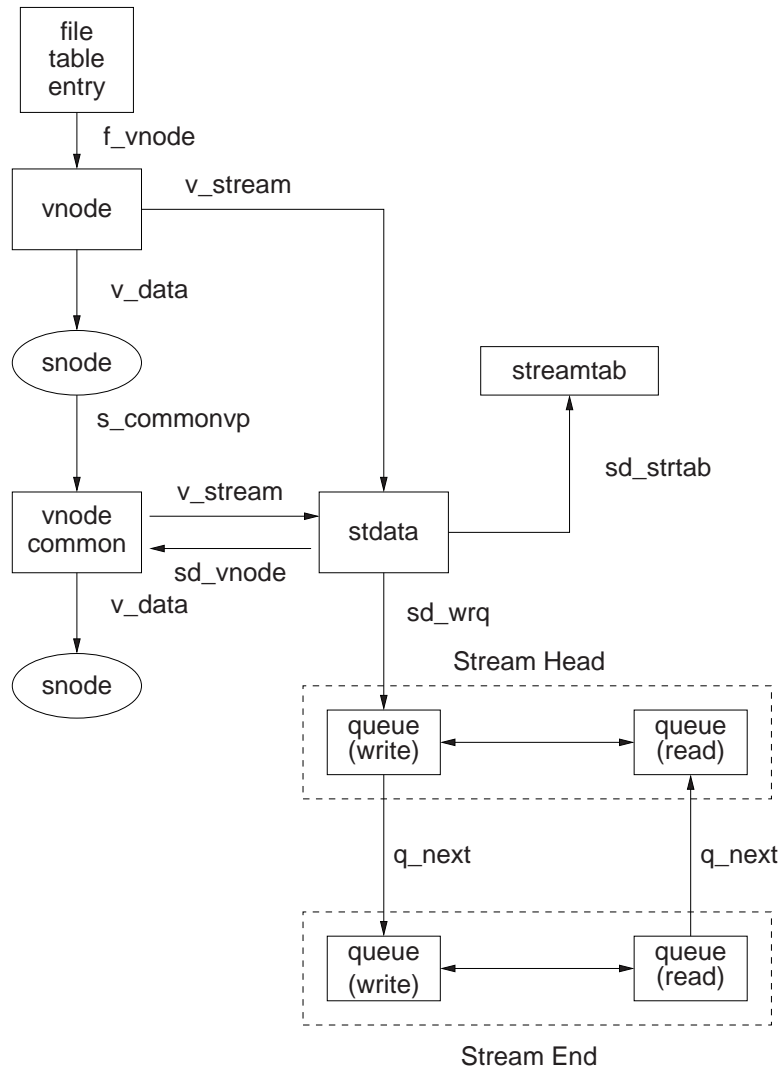
```
struct streamtab {
    struct qinit *st_rdinit;
    struct qinit *st_wrinit;
    struct qinit *st_muxrinit;
    struct qinit *st_muxwinit;
};
```

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

If the `open` call is the initial file open, a *Stream* is created. (There is one *Stream* per major/minor device pair.) First, an entry is allocated in the user's file table and a *vnode* is created to represent the opened file. The file table entry is initialized to point to the allocated *vnode* (see *f_vnode* in Figure 3.3) and the *vnode* is initialized to specify a file of type character special.

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

The *snode* contains the file system dependent information. It is associated with the *vnode* representing the device. The `s_commonvp` field of the *snode* points to the common device *vnode*. The *vnode* field, `v_data`, contains a pointer to the *snode*. Instead of maintaining a pointer to the *vnode*, the *snode* contains the *vnode* as an element. The `sd_vnode` field of `stdata` is initialized to point to the allocated *vnode*. The `v_stream` field of the *vnode* data structure is initialized to point to the *Stream header*, thus there is a forward and backward pointer between the *Stream header* and the *vnode*. There is one *Stream header* per *Stream*. The header is used by *STREAMS* while performing operations on the *Stream*. In the downstream portion of the *Stream*, the *Stream header* points to the downstream half of the *Stream head* queue pair. Similarly, the upstream portion of the *Stream* terminates at the *Stream header*, since the upstream half of the *Stream head* queue pair points to the header. As shown in Figure 3.3, from the *Stream header* onward, a *Stream* is constructed of linked queue pairs.

Figure 3.3: *Opened STREAMS-based Driver*

Next, a **queue** structure pair is allocated for the driver. The queue limits are initialized to those values specified in the corresponding **module_info** structure. The queue processing routines are initialized to those specified by the corresponding **qinit** structure.

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

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

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

3.2.2 Creating a STREAMS-based Pipe

In addition to opening a *STREAMS*-based driver, a *Stream* can be created by creating a pipe [see `pipe(2)`]. Since pipes are not character devices, *STREAMS* creates and initializes a `streamtab(9)` structure for each end of the pipe. As with modules and drivers, the `streamtab` structure defines the pipe. The `st_rdinit`, however, points to the read `qinit` structure for the *Stream head* and not for a driver. Similarly, the `st_wdinit` points to the *Stream head*'s write `qinit` structure and not to a driver. The `st_muxrinit` and `st_muxwinit` are initialized to null since a pipe cannot be a multiplexor driver.

When the pipe system call is executed, two *Streams* are created. *STREAMS* follows the procedures similar to those of opening a driver; however, duplicate data structures are created. That is, two entries are allocated in the user's file table and two `vnodes` are created to represent each end of the pipe, as shown in Figure 3.4. The file table entries are initialized to point to the allocated `vnodes` and each `vnode` is initialized to specify a file of type *FIFO*.

Next, two *Stream headers* are created from `stdata` data structures and two *Stream heads* are created from two pairs of `queue` structures. The content of `stdata` and `queue` are initialized with the same values for all pipes.

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

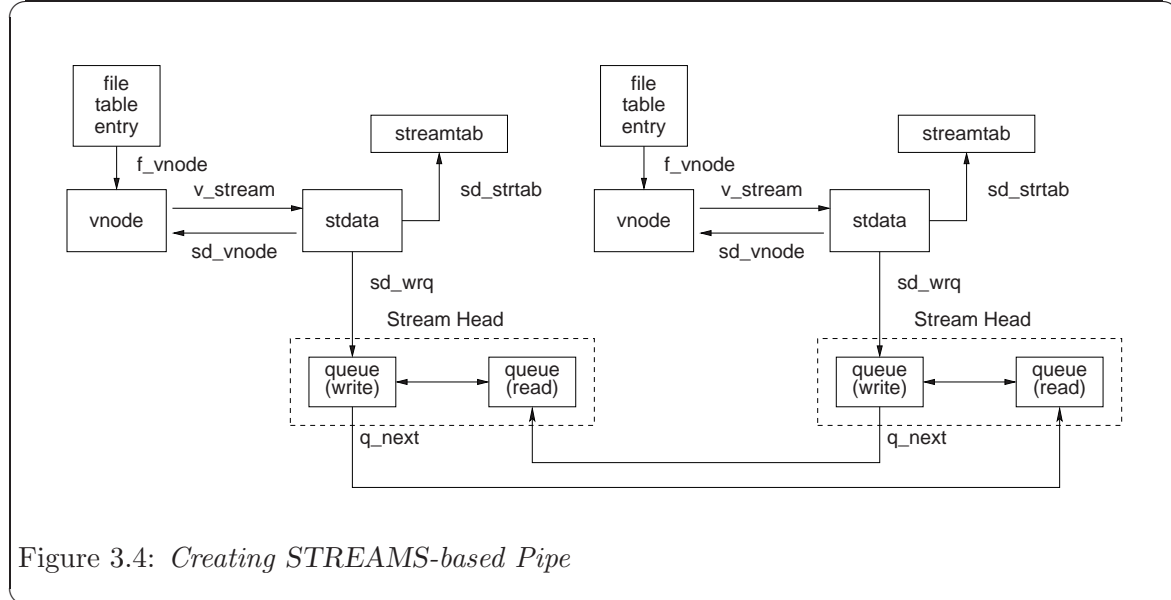


Figure 3.4: Creating STREAMS-based Pipe

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

3.2.3 Adding and Removing Modules

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

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

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

Each push of a module is independent, even in the same *Stream*. If the same module is pushed more than once on a *Stream*, there will be multiple occurrences of that module in the *Stream*. The total number of pushable modules that may be contained on any one *Stream* is limited by the kernel parameter `NSTRPUSH` (see [Appendix E \[STREAMS Configuration\]](#), [page 323](#)).

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

3.2.4 Closing the Stream

The last close to a *STREAMS* file dismantles the *Stream*. Dismantling consists of popping any modules on the *Stream* and closing the driver. Before a module is popped, the close may delay to allow any messages on the write message queue of the module to be drained by module processing. Similarly, before the driver is closed, the close may delay to allow any messages on the write message queue of the driver to be drained by driver processing. If `O_NDELAY` (or `O_NONBLOCK`) [see `open(2)`] is clear, close will wait up to '15' seconds for each module to drain and up to '15' seconds for the driver to drain. If `O_NDELAY` (or `O_NONBLOCK`) is set, the pop is performed immediately and the driver is closed without delay. Messages can remain queued, for example, if flow control is inhibiting execution of the write queue service procedure. When all modules are popped and any wait for the driver to drain is completed, the driver `close` routine is called. On return from the driver `close`, any messages left on the driver's queues are freed, and the `queue` and `stdata` structures are deallocated.

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

Finally, the user's *file table* entry and the *vnode* are deallocated and the *file* is closed.

3.2.5 Stream Construction Example

The following example extends the previous communications device echoing example (see [\[undefined\]](#) [\[undefined\]](#), page [\[undefined\]](#)) by inserting a module in the Stream. The (hypothetical) module in this example can convert (change case, delete, duplicate) selected alphabetic characters.

3.2.5.1 Inserting Modules

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

```
#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    BUFLEN    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 */

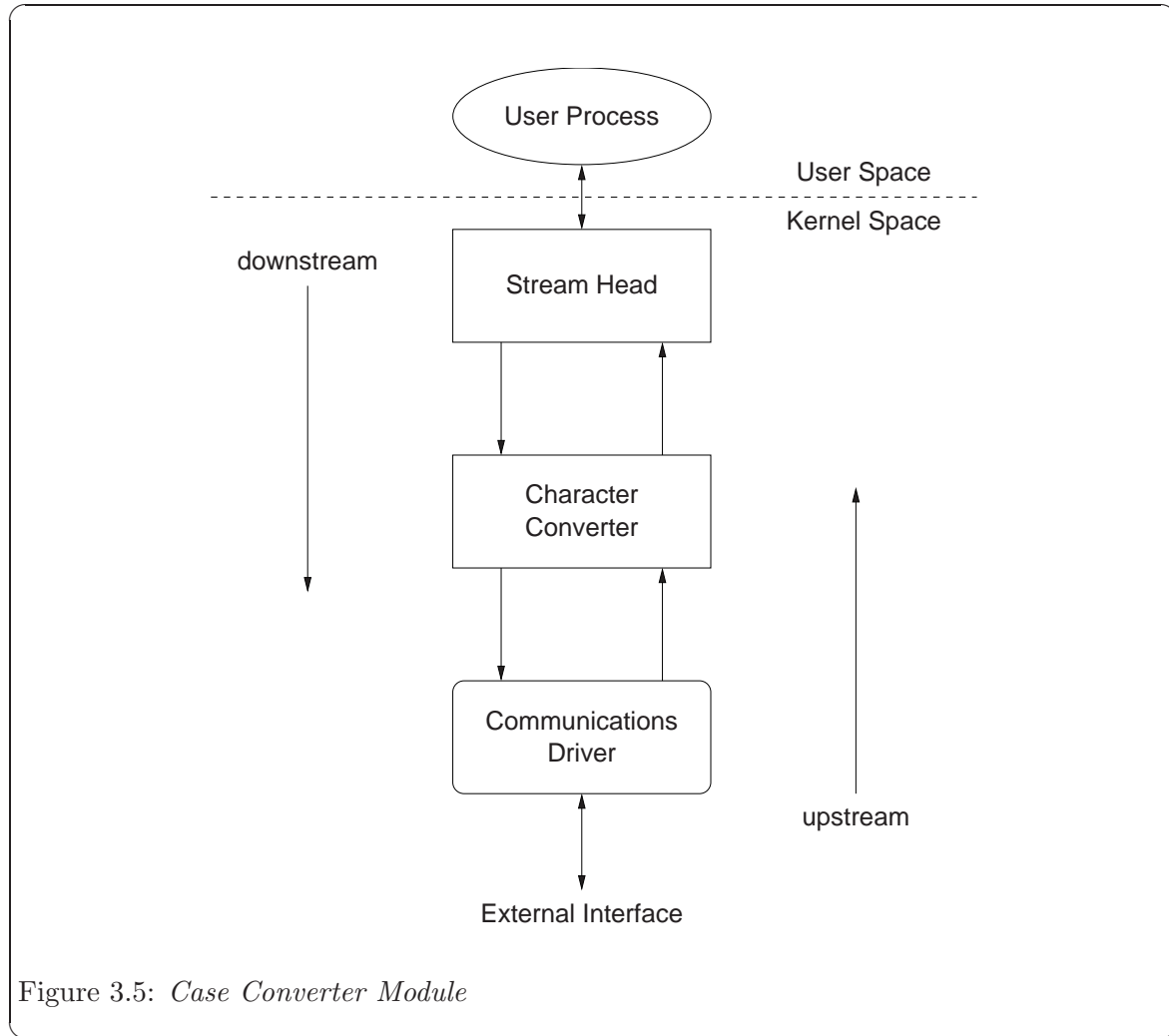
main()
{
    char buf[BUFLEN];
    int fd, count;
    struct strioctl1 strioctl;
```

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

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

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

The `I_PUSH ioctl` call directs the *Stream* head to insert the character conversion module between the driver and the *Stream* head, creating the *Stream* shown in Figure 3.5. As with drivers, this module resides in the kernel and must have been configured into the system before it was booted.



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

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

3.2.5.2 Module and Driver Control

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

```
/* change all uppercase vowels to lowercase */
striocctl.ic_cmd = XCASE;
striocctl.ic_timeout = 0;          /* default timeout (15 sec) */
striocctl.ic_dp = "AEIOU";
striocctl.ic_len = strlen(striocctl.ic_dp);

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

/* delete all instances of the chars 'x' and 'X' */
striocctl.ic_cmd = DELETE;
striocctl.ic_dp = "xX";
striocctl.ic_len = strlen(striocctl.ic_dp);

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

`ioctl` requests are issued to *STREAMS* drivers and modules indirectly, using the `I_STR` `ioctl` call [see `streamio(7)`]. The argument to `I_STR` must be a pointer to a `striocctl` structure, which specifies the request to be made to a module or driver. This structure is defined in ‘`sys/stropts.h`’ and 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 */
};
```

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

`I_STR` is intercepted by the *Stream head*, which packages it into a message, using information contained in the `striocctl` structure, and sends the message downstream. Any module that

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

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

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

The remainder of this example is identical to the example in [Chapter 7 \[Overview of STREAMS Modules and Drivers\]](#), page 105:

```

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

```

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

The `exit` system call will dismantle the *Stream* before terminating the process. The character conversion module will be removed from the *Stream* automatically when it is closed. Alternatively, modules may be removed from a *Stream* using the `I_POP ioctl` call described in `streamio(7)`. This call removes the topmost module on the *Stream*, and enables a user process to alter the configuration of a *Stream* dynamically, by popping modules as needed. A few of the important `ioctl` requests supported by *STREAMS* have been discussed. Several other requests are available to support operations such as determining if a given module exists on the *Stream*, or flushing the data on a *Stream*. These requests are described fully in `streamio(7)`.

4 STREAMS Processing Routines

4.1 STREAMS Put and Service Procedures

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

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

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

4.1.1 Put Procedure

A **put** procedure is the queue routine that receives messages from the preceding queues in the *Stream*. Messages are passed between queues by a procedure in one queue calling the **put** procedure contained in the following queue. A call to the **put** procedure in the appropriate direction is generally the only way to pass messages between *STREAMS* components. There is usually a separate **put** procedure for the read and write queues because of the full-duplex operation of most *Streams*. However, there can be a single **put** procedure shared between both the read and write queues.

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

Each *STREAMS* component accesses the adjacent **put** procedure as a subroutine. For example, consider that 'modA', 'modB', and 'modC' are three consecutive components in a *Stream*, with 'modC' connected to the *Stream head*. If 'modA' receives a message to be sent upstream, 'modA' processes that message and calls 'modB's read **put** procedure, which processes it and calls 'modC's read **put** procedure, which processes it and calls the *Stream head's* read **put** procedure. Thus, the message will be passed along the *Stream* in one continuous processing sequence. This sequence has the benefit of completing the entire processing in a short time with low overhead (subroutine calls). On the other hand, if this sequence is lengthy and the processing is implemented on a multi-user system, then this manner of processing may be good for this *Stream* but may be detrimental for others. *Streams* may have to wait too long to get their turn, since each **put** procedure is called from the preceding one, and the kernel stack (or interrupt stack) grows with each function call. The possibility of running off the stack exists, thus panicking the system or producing undeterminate results.

4.1.2 Service Procedure

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

The *STREAMS* scheduler is separate and distinct from the *UNIX* system process scheduler. It is concerned only with queues linked on the *STREAMS* scheduling queue. The scheduler calls each **service** procedure of the scheduled queues one at a time in a *First-In-First-Out (FIFO)* manner.

The scheduling of queue **service** routines is machine dependent. However, they are guaranteed to run before returning to user level.

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

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

Rules that should be observed in **put** and **service** procedures are listed in [Chapter 7 \[Overview of STREAMS Modules and Drivers\]](#), page 105.

4.2 An Asynchronous Stream Example

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

Communications software support for these terminals is provided via a *STREAMS* based asynchronous protocol. The protocol includes a variety of options that are set when a

terminal operator dials in to log on. The options are determined by a *STREAMS* user process, `getstrm`, which analyzes data sent to it through a series of dialogs (prompts and responses) between the process and terminal operator.

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

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

These options are set with the following modules:

CHARPROC

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

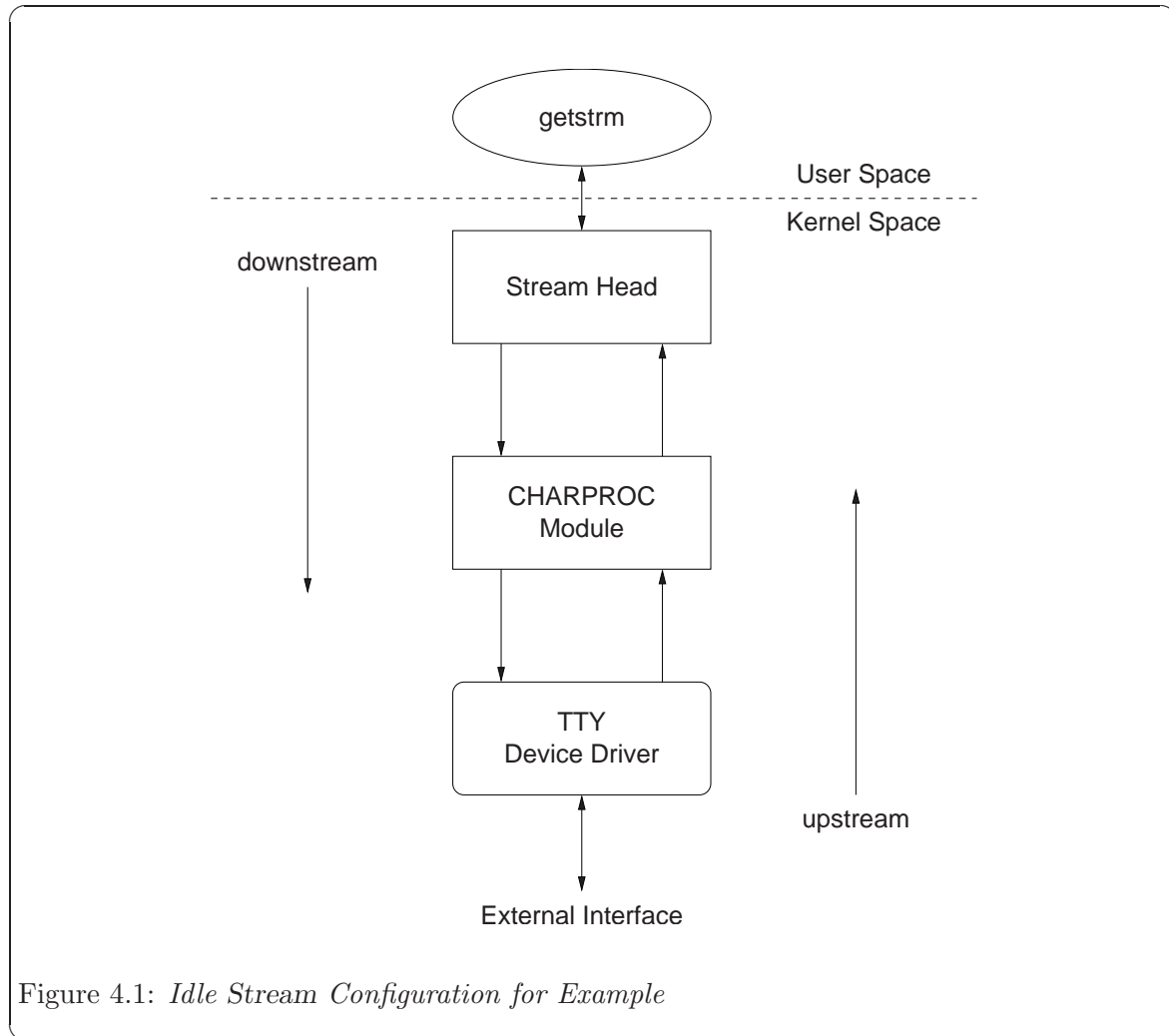
CANONPROC

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

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

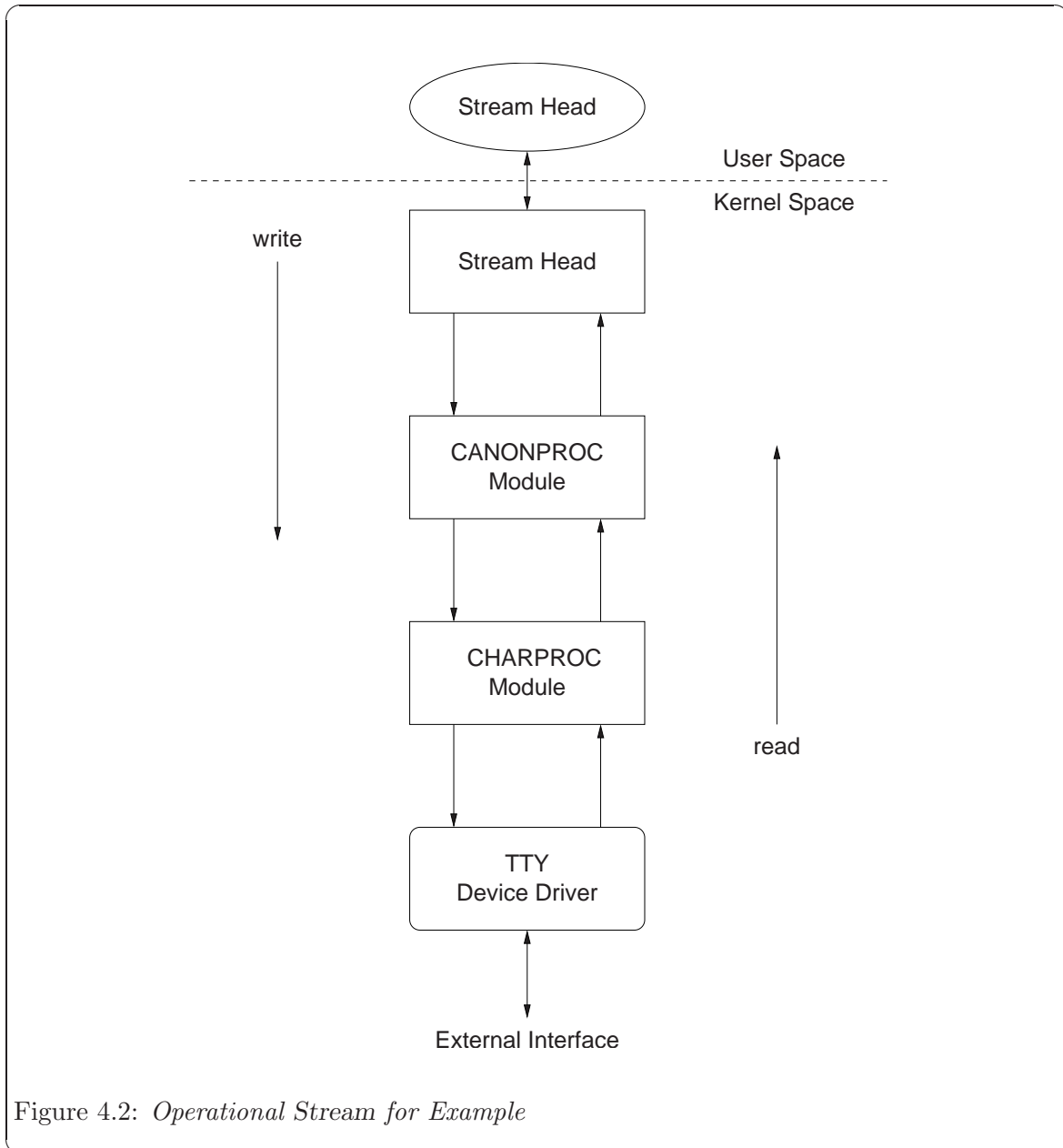
At system initialization a user process, `getstrm`, is created for each *tty* port. `getstrm` opens a *Stream* to its port and pushes the *CHARPROC* module onto the *Stream* by use of an `ioctl I_PUSH` command. Then, the process issues a `getmsg` system call to the *Stream* and sleeps until a message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

The initial idle *Stream*, shown in [Figure 4.1](#), contains only one pushable module, *CHARPROC*. The device driver is a limited function raw *tty* driver connected to a limited-function communication port. The driver and port transparently transmit and receive one unbuffered character at a time.



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

Figure 4.2 continues the example and associates kernel operations with user-level system calls. As a result of initializing operations and pushing a module, the *Stream* for port one has the following configuration:



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

4.2.1 Read-Side Processing

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

4.2.1.1 Driver Processing

The user process has been blocked on the `getmsg(2)` system call while waiting for a message to reach the *Stream head*, and the device driver independently waits for input of a character from the port hardware or for a message from upstream. Upon receipt of an input character interrupt from the port, the driver places the associated character in an `M_DATA` message, allocated previously. Then, the driver sends the message to the `CHARPROC` module by calling `CHARPROC`'s upstream `put` procedure. On return from `CHARPROC`, the driver calls the `allocb(9)` utility routine to get another message for the next character.

4.2.1.2 CHARPROC

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

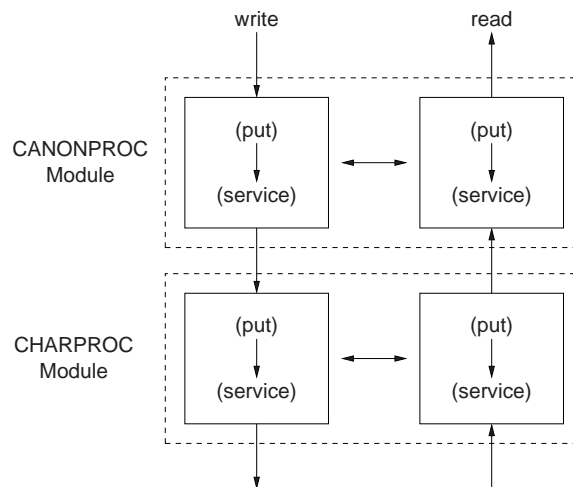


Figure 4.3: *Module Put and Service Procedures*

When the driver calls `CHARPROC`'s read queue `put` procedure, the procedure checks private data flags in the queue. In this case, the flags indicate that echoing is to be performed (recall that echoing is optional and that we are working with port hardware which can not automatically echo). `CHARPROC` causes the echo to be transmitted back to the terminal by first making a copy of the message with a *STREAMS* utility routine. Then, `CHARPROC` uses another utility routine to obtain the address of its own write queue. Finally, the `CHARPROC` read `put` procedure calls its write `put` procedure and passes it the message copy. The write procedure sends the message to the driver to effect the echo and then returns to the read procedure.

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

write procedure) to the driver. This manner of processing is intended to produce the character echo in a minimal time frame.

After returning from echo processing, the **CHARPROC** read **put** procedure checks another of its private data flags and determines that parity checking should be performed on the input character. Parity should most reasonably be checked as part of echo processing. However, for this example, parity is checked only when the characters are sent upstream. This relaxes the timing in which the checking must occur, that is, it can be deferred along with the canonical processing. **CHARPROC** uses **putq(9)** to schedule the (original) message for parity check processing by its read **service** procedure. When the **CHARPROC** read **service** procedure is complete, it forwards the message to the read **put** procedure of **CANONPROC**. Note that if parity checking was not required, the **CHARPROC** **put** procedure would call the **CANONPROC** **put** procedure directly.

4.2.1.3 CANONPROC

CANONPROC performs canonical processing. As implemented, all read queue processing is performed in its **service** procedure so that **CANONPROC**'s **put** procedure simply calls **putq(9)** to schedule the message for its read **service** procedure and then exits. The **service** procedure extracts the character from the message buffer and places it in the "line buffer" contained in another **M_DATA** message it is constructing. Then, the message which contained the single character is returned to the buffer pool. If the character received was not an *end-of-line*, **CANONPROC** exits. Otherwise, a complete line has been assembled and **CANONPROC** sends the message upstream to the *Stream head* which unblocks the user process from the **getmsg(2)** call and passes it the contents of the message.

4.2.2 Write-Side Processing

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

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

For terminal output, it is presumed that **M_DATA** messages, sent by **write(2)** system calls, contain multiple characters. In general, *STREAMS* returns to the user process immediately after processing the **write** call so that the process may send additional messages. Flow control will eventually block the sending process. The messages can queue on the write-side of the driver because of character transmission timing. When a message is received by the driver's write **put** procedure, the procedure will use **putq(9)** to place the message on its write-side service message queue if the driver is currently transmitting a previous message buffer. However, there is generally no write queue **service** procedure in a device driver. Driver output interrupt processing takes the place of scheduling and performs the **service** procedure functions, removing messages from the queue.

4.2.3 Analysis

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

5 STREAMS Messages

5.1 STREAMS Messages Overview

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

5.1.1 Message Types

There are several different *STREAMS* messages (see [Section B.1 \[Message Types\]](#), page 281) and they are defined in `'sys/stream.h'`. The messages differ in their intended purpose and their queueing priority. The contents of certain message types can be transferred between a process and a *Stream* by use of system calls.

Below, the message types are briefly described and classified according to their queueing priority.

Ordinary Messages (also called normal messages):

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

High Priority Messages:

M_COPYIN	-	U	Copy in data for transparent <code>ioctl</code> s, sent upstream
M_COPYOUT	-	U	Copy out data for transparent <code>ioctl</code> s, sent upstream
M_ERROR	-	U	Report downstream error condition, sent upstream
M_FLUSH	D	U	Flush module queue
M_HANGUP	-	U	Set a <i>Stream head</i> hangup condition, sent upstream
M_IOCACK	-	U	Positive <code>ioctl(2)</code> acknowledgement
M_IOCDATA	D	-	Data for transparent <code>ioctl</code> s, sent downstream
M_IOCNAK	-	U	Negative <code>ioctl(2)</code> acknowledgement
M_PCPROTO	D	U	Protocol control information
M_PCRSE	D	U	Reserved for internal use
M_PCSIG	-	U	Signal sent from a module/driver to a user
M_READ	D	-	Read notification, sent downstream

M_STOP	D	-	Suspend output
M_START	D	-	Restart stopped device output
M_STOPI	D	-	Suspend input
M_STARTI	D	-	Restart stopped device input

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

5.1.2 Expedited Data

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

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

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

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

Drivers and modules define separate high and low water marks for priority band data flow. (Water marks are defined for each queue and they indicate the upper and lower limit of bytes that can be contained on the queue; see `M_SETOPTS` in see [Section B.1 \[Message Types\]](#), [page 281](#)). The default water marks for priority band data and normal data are the same. The *Stream* head also ensures that incoming priority band data are not blocked by normal data already on the queue. This is accomplished by associating a priority with the messages. This priority implies a certain ordering of the messages in the queue. (Message queues and priorities are discussed later in this chapter.)

Within the *STREAMS* mechanism and in this guide expedited data are also referred to as priority band data.

5.2 STREAMS Message Structure

All messages are composed of one or more message blocks. A message block is a linked triplet of two structures and a variable length data buffer. The structures are a message block (`msgb`) and a data block (`datab`). The data buffer is a location in memory where the data of a message are stored.

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

```

    struct datab *b_datap;      /* data block */
    unsigned char b_band;       /* message priority */
    unsigned char b_pad1;
    unsigned short b_flag;      /* see below - message flags */
    long b_pad2;
};

typedef struct msgb mblk_t;

/* message flags.  these are interpreted by the stream head. */
#define msgmark    0x01        /* last byte of message is "marked" */
#define msgnolop   0x02        /* don't loop message around to write
                                side of stream */
#define msgdelim   0x04        /* message is delimited */
#define msgnoget   0x08        /* getq does not return message */
#define msgatten   0x20        /* attention to on read side */

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

struct datab {
    union {
        struct datab *freep;
        struct free_rtn *frtnp;
    } db_f;          /* used internally */
    unsigned char *db_base;    /* first byte of buffer */
    unsigned char *db_lim;     /* last byte+1 of buffer */
    unsigned char db_ref;      /* message count pointing to this block
                                */
    unsigned char db_type;     /* message type */
    unsigned char db_iswhat;   /* status of message/dat buffer triplet
                                */
    unsigned int db_size;      /* used internally */
    caddr_t db_msgaddr; /* triplet message header pointing to datab */
    long db_filler;          /* reserved for future use */
};

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

typedef struct datab dblk_t;
typedef struct free_rtn frtn_t;

```

The field *b_band* determines where the message is placed when it is enqueued using the *STREAMS* utility routines. This field has no meaning for high priority messages and is set to zero for these messages. When a message is allocated via `allocb(9)`, the *b_band* field will be initially set to zero. Modules and drivers may set this field if so desired.

5.2.1 Message Linkage

The message block is used to link messages on a message queue, link message blocks to form a message, and manage the reading and writing of the associated data buffer. The *b_rptr* and *b_wptr* fields in the *msgb* structure are used to locate the data currently contained in

the buffer. As shown in [Figure 5.1](#), the message block (`mblk_t`) points to the data block of the triplet. The data block contains the message type, buffer limits, and control variables. *STREAMS* allocates message buffer blocks of varying sizes. `db_base` and `db_lim` are the fixed beginning and end (+1) of the buffer.

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

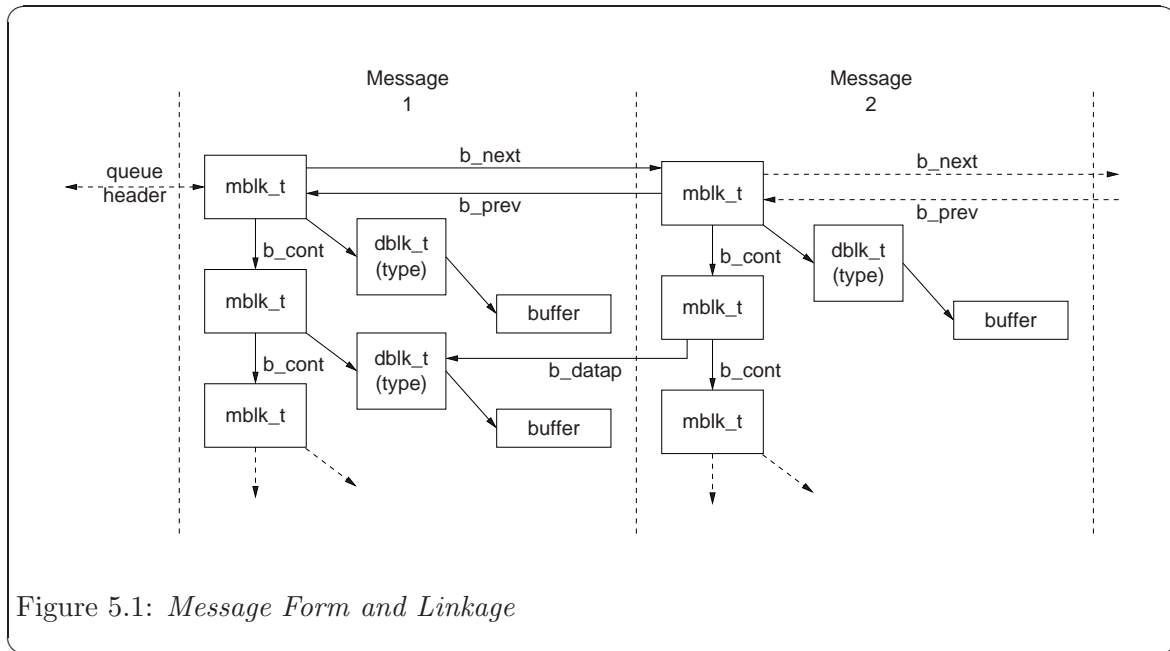


Figure 5.1: *Message Form and Linkage*

A message may occur singly, as when it is processed by a `put` procedure, or it may be linked on the message queue in a `queue`, generally waiting to be processed by the service procedure. Message ‘2’, as shown in [Figure 5.1](#), links to message ‘1’.

Note that a data block in message ‘1’ is shared between message ‘1’ and another message. Multiple message blocks can point to the same data block to conserve storage and to avoid copying overhead. For example, the same data block, with associated buffer, may be referenced in two messages, from separate modules that implement separate protocol levels. [Figure 5.1](#), illustrates the concept, but data blocks would not typically be shared by messages on the same queue). The buffer can be retransmitted, if required because of errors or timeouts, from either protocol level without replicating the data. Data block sharing is accomplished by means of a utility routine [see `dupmsg(9)` or [Appendix C \[STREAMS Utilities\]](#), page 295]. *STREAMS* maintains a count of the message blocks sharing a data block in the `db_ref` field.

STREAMS provides utility routines and macros, specified in [Appendix C \[STREAMS Utilities\]](#), page 295, to assist in managing messages and message queues, and to assist in other areas of module and driver development. A utility routine should always be used when

operating on a message queue or accessing the message storage pool. If messages are manipulated on the queue without using the *STREAMS* utilities, the message ordering may become confused and lead to inconsistent results.

5.2.2 Sending and Receiving Messages

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

Modules and drivers do not interact directly with any system calls except `open(2)` and `close(2)`. The *Stream head* handles all message translation and passing between user processes and *STREAMS* components. Message transfer between processes and the *Stream head* can occur in different forms. For example, `M_DATA` and `M_PROTO` messages can be transferred in their direct form by the `getmsg(2)` and `putmsg(2)` system calls. Alternatively, `write(2)` causes one or more `M_DATA` messages to be created from the data buffer supplied in the call. `M_DATA` messages received at the *Stream head* will be consumed by `read(2)` and copied into the user buffer. As another example, `M_SIG` causes the *Stream head* to send a signal to a process.

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

STREAMS enables modules to create messages and pass them to neighboring modules. However, the `read(2)` and `write(2)` system calls are not sufficient to enable a user process to generate and receive all such messages. First, read and write are byte-stream oriented with no concept of message boundaries. To support service interfaces, the message boundary of each service primitive must be preserved so that the beginning and end of each primitive can be located. Also, read and write offer only one buffer to the user for transmitting and receiving *STREAMS* messages. If control information and data were placed in a single buffer, the user would have to parse the contents of the buffer to separate the data from the control information.

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

The format of `putmsg` is as follows:

```
int
putmsg(fd, ctlptr, dataptr, flags)
```



```
int fd, flags;
struct strbuf *ctlptr, *dataptr;
```

fd identifies the *Stream* to which the message will be passed, *ctlptr* and *dataptr* identify the control and data parts of the message, and *flags* may be used to specify that a high priority message (M_PCPROTO) should be sent. When a control part is present, setting *flags* to '0' generates an M_PROTO message. If *flags* is set to RS_HIPRI, an M_PCPROTO message is generated.

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

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

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

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

The `getmsg` system call retrieves M_DATA, M_PROTO, or M_PCPROTO messages available at the *Stream* head, and has the following format:

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

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

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

`putpmsg()` has the following interface:

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

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

The call

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

is equivalent to the the system call

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

and the call


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

is equivalent to the system call

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

If `MSG_HIPRI` is set and `band` is non-zero, `putpmsg()` fails with `[EINVAL]`.

`getpmsg()` has the following format:

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

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

The call

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

is equivalent to

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

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

5.2.3 Control of Stream Head Processing

The `M_SETOPTS` message allows a driver or module to exercise control over certain *Stream head* processing. An `M_SETOPTS` can be sent upstream at any time. The *Stream head* responds to the message by altering the processing associated with certain system calls. The options to be modified are specified by the contents of the `stroptions` structure (see [Appendix A \[STREAMS Data Structures\], page 275](#)) contained in the message.

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

5.2.3.1 Read Options

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

byte-stream (`RNORM`)

The `read(2)` call completes when the byte count is satisfied, the *Stream head* read queue becomes empty, or a zero length message is encountered. In the last

case, the zero length message is put back on the queue. A subsequent read will return '0' bytes.

message non-discard (RMSGN)

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

message discard (RMSGD)

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

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

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

normal protocol (RPROTNORM)

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

protocol discard (RPROTDIS)

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

protocol data (RPROTDAT)

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

5.2.3.2 Write Options

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

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

The write offset value must be smaller than the maximum *STREAMS* message size, `STRMSGSZ` (see [\[undefined\]](#) [\[undefined\]](#), page [\[undefined\]](#)). In certain cases (e.g., if a buffer large enough to hold the '`offset+data`' is not currently available), the write offset might not be included in the block. To handle all possibilities, modules and drivers should not assume that the offset exists in a message, but should always check the message.

The intended use of write offset is to leave room for a module or a driver to place a protocol header before user data in the message rather than by allocating and prepending a separate message.

5.3 STREAMS Message Queues and Priority

Message queues grow when the *STREAMS* scheduler is delayed from calling a service procedure because of system activity, or when the procedure is blocked by flow control. When

called by the scheduler the service procedure processes enqueued messages in a *First-In-First-Out (FIFO)* manner. However, expedited data support and certain conditions require that associated messages (e.g., an `M_ERROR`) reach their *Stream* destination as rapidly as possible. This is accomplished by associating priorities to the messages. These priorities imply a certain ordering of messages on the queue as shown in Figure 5.2. Each message has a priority band associated with it. Ordinary messages have a priority of zero. High priority messages are high priority by nature of their message type. Their priority band is ignored. By convention, they are not affected by flow control. The `putq(9)` utility routine places high priority messages at the head of the message queue followed by priority band messages (expedited data) and ordinary messages.

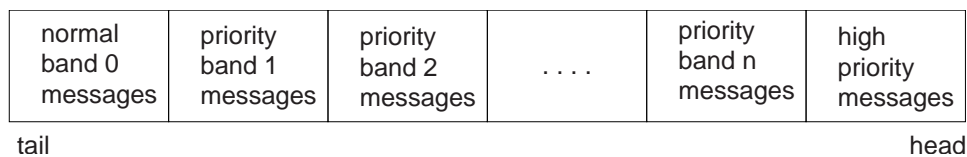


Figure 5.2: *Message Ordering on a Queue*

When a message is queued, it is placed after the messages of the same priority already on the queue (i.e., *FIFO* within their order of queueing). This affects the flow control parameters associated with the band of the same priority. Message priorities range from ‘0’ (normal) to ‘255’ (highest). This provides up to ‘256’ bands of message flow within a *Stream*. Expedited data can be implemented with one extra band of flow (priority band 1) of data. This is shown in Figure 5.3.

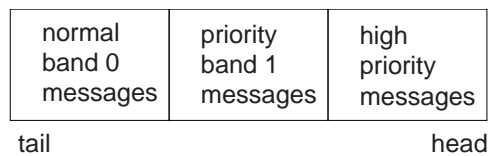


Figure 5.3: *Message Ordering with One Priority Band*

High priority messages are not subject to flow control. When they are queued by `putq(9)`, the associated queue is always scheduled (in the same manner as any queue; following all other queues currently scheduled). When the service procedure is called by the scheduler, the procedure uses `getq(9)` to retrieve the first message on queue, which will be a high priority message, if present. Service procedures must be implemented to act on high priority messages immediately. The above mechanisms priority message queueing, absence of flow control, and immediate processing by a procedure result in rapid transport of high priority messages between the originating and destination components in the *Stream*.

Several routines are provided to aid users in controlling each priority band of data flow. These routines are `flushband(9)`, `bcanput(9)`, `strqget(9)`, and `strqset(9)`. The `flushband()` routine is discussed in [Section 7.3 \[STREAMS Flush Handling\]](#), [page 124](#), the `bcanput()` routine is discussed under [\[Message Processing\]](#), [page 72](#), later in this chapter, and the other two routines are described next. [Appendix C \[STREAMS Utilities\]](#), [page 295](#), also has a description of these routines.

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

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

The information is returned in the `long` referenced by `valp`. The fields that can be obtained are defined by the following:

```
typedef enum qfields {
    QHIWAT = 0,      /* q_hiwat or qb_hiwat */
    QLOWAT = 1,      /* q_lowat or qb_lowat */
    QMAXPSZ = 2,     /* q_maxpsz */
    QMINPSZ = 3,     /* q_minpsz */
    QCOUNT = 4,     /* q_count or qb_count */
    QFIRST = 5,      /* q_first or qb_first */
    QLAST = 6,       /* q_last or qb_last */
    QFLAG = 7,       /* q_flag or qb_flag */
    QBAD = 8
} qfields_t;
```

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

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

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

The updated information is provided by `val`. `strqset()` returns ‘0’ on success and an error number on failure. If the field is intended to be read-only, then the error `[EPERM]` is returned and the field is left unchanged. The following fields are currently read-only: `QCOUNT`, `QFIRST`, `QLAST`, and `QFLAG`.

The `ioctl`s `I_FLUSHBAND`, `I_CKBAND`, `I_GETBAND`, `I_CANPUT`, and `I_ATMARK` support multiple bands of data flow. The `ioctl` `I_FLUSHBAND` allows a user to flush a particular band of messages. It is discussed in more detail in [Section 7.3 \[STREAMS Flush Handling\]](#), [page 124](#).

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

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

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

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

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

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

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

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

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

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

The `ioctl` `I_ATMARK` has the following format:

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

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

5.3.1 The queue Structure

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

The operation of a queue revolves around the `queue` structure:

```
struct queue {
    struct qinit *q_qinfo;    /* procedures and limits for queue */
    struct msgb *q_first;    /* head of message queue for this queue */
    struct msgb *q_last;    /* tail of message queue for this queue */
    struct queue *q_next;    /* next queue in Stream */
    struct queue *q_link;    /* to next queue for scheduling */
    _VOID *q_ptr;            /* to private data structure */
    ulong q_count;           /* number of bytes in queue */
    ulong q_flag;            /* queue state */
};
```

```

    long q_minpsz;          /* min packet size accepted by this module */
    long q_maxpsz;          /* max packet size accepted by this module */
    ulong q_hiwat;          /* queue high water mark for flow control */
    ulong q_lowat;          /* queue low water mark for flow control */
    struct qband *q_bandp;  /* separate flow information */
    unsigned char q_nband;  /* number of priority bands */
    unsigned char q_blocked; /* number of bands flow controlled */
    unsigned char q_pad1[2]; /* reserved for future use */
    long q_pad2[2];         /* reserved for future use */
};

typedef struct queue queue_t;

```

Queues are always allocated in pairs (read and write); one queue pair per a module, a driver, or a *Stream head*. A queue contains a linked list of messages. When a queue pair is allocated, the following fields are initialized by *STREAMS*:

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

Copying values from *module_info* allows them to be changed in the queue without modifying the *streamtab* and *module_info* values.

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

5.3.1.1 Using queue Information

Modules and drivers should use *STREAMS* utility routines (see [Appendix C \[STREAMS Utilities\]](#), page 295) to alter *q_first*, *q_last*, *q_count*, and *q_flag*.

Modules and drivers can change *q_ptr*, *q_minpsz*, *q_maxpsz*, *q_hiwat*, and *q_lowat*.

Modules and drivers can read but should not change *q_qinfo*, *q_next*, *q_bandp*, and *q_nband*.

Modules and drivers should not touch *q_link*, *q_pad1*, and *q_pad2*.

5.3.1.2 Queue Flags

Programmers using the *STREAMS* mechanism should be aware of the following queue flags:

QENAB	queue is enabled to run the service procedure (it is on the run queue)
QWANTR	someone wants to read from the queue
QWANTW	someone wants to write to the queue
QFULL	queue is full
QREADR	set for read queues
QUSE	queue has been allocated
QNOENB	do not enable the queue when data are placed on it
QBACK	queue has been back-enabled
QOLD	queue supports module/driver interface to open/close developed prior to <i>UNIX System V Release 4.0</i>
QHLIST	the <i>Stream head</i> write queue is scanned

5.3.1.3 The qband Structure

The queue flow information for each band is contained in a `qband` structure. It is defined as follows:

```

struct qband {
    struct qband *qb_next;    /* next band's info */
    ulong qb_count;          /* number of bytes in band */
    struct msgb *qb_first;    /* beginning of band's data */
    struct msgb *qb_last;    /* end of band's data */
    ulong qb_hiwat;          /* high water mark for band */
    ulong qb_lowat;          /* low water mark for band */
    ulong qb_flag;           /* flag, QB_FULL, denotes that a band
                             of data flow is flow controlled */
    long qb_pad1;            /* reserved for future use */
};

typedef struct qband qband_t;

/*
 * qband flags
 */
#define QB_FULL      0x01    /* band is considered full */
#define QB_WANTW     0x02    /* someone wants to write to band */
#define QB_BACK      0x04    /* queue has been back-enabled */

```

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

Each band has a separate high and low water mark, `qb_hiwat` and `qb_lowat`. These are initially set to the queue's `q_hiwat` and `q_lowat` respectively. Modules and drivers may change these values if desired through the `strqset(9)` function. Three flags, `QB_FULL`, `QB_WANTW`, and `QB_BACK`, are defined for `qb_flag`. `QB_FULL` denotes that the particular band is full. `QB_WANTW` indicates that someone tried to write to the band that was flow controlled. `QB_BACK` is set when the service procedure runs as a result of being back-enabled because the queue is no longer flow-controlled.

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

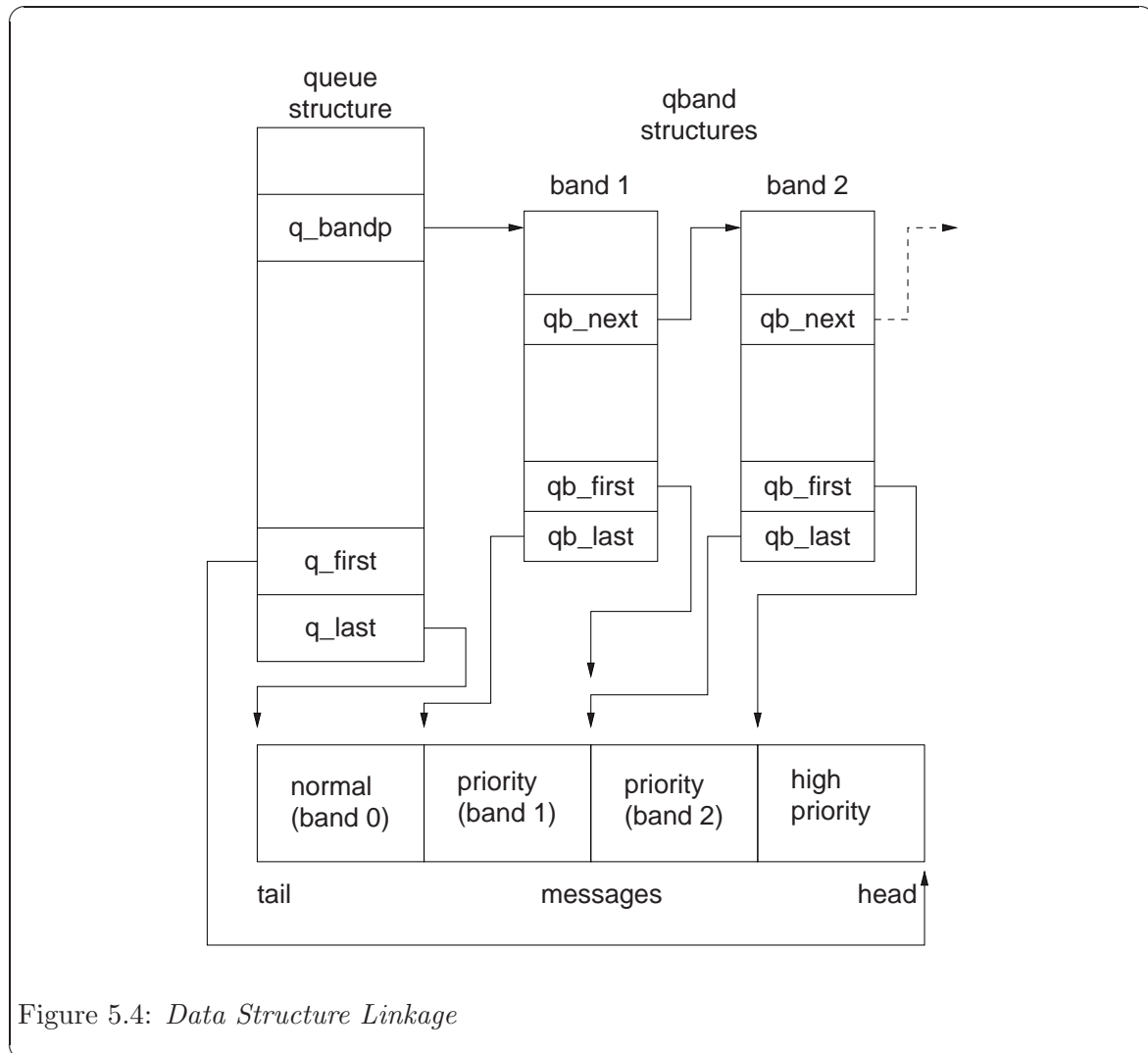
5.3.1.4 Using qband Information

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

Drivers and modules are allowed to change the *qp-hiwait* and *qp-lowat* fields of the **qband** structure.

Drivers and modules may only read the *qb-count*, *qb-first*, *qb-last*, and *qb-flag* fields of the **qband** structure.

The *pad* fields should not be used in the **qband** structure; they are intended for future use. The following figure depicts a queue with two priority bands of flow.



5.3.2 Message Processing

Put procedures are generally required in pushable modules. Service procedures are optional. If the **put** routine enqueues messages, there must exist a corresponding **service** routine that handles the enqueued messages. If the **put** routine does not enqueue messages, the **service** routine need not exist.

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

1. A message is received by the **put** procedure in a queue, where some processing may be performed on the message.
2. The **put** procedure places the message on the queue by use of the **putq(9)** utility routine for the **service** procedure to perform further processing at some later time.
3. **putq(9)** places the message on the queue based on its priority.
4. Then, **putq(9)** makes the queue ready for execution by the *STREAMS* scheduler following all other queues currently scheduled.
5. After some indeterminate delay (intended to be short), the *STREAMS* scheduler calls the **service** procedure.
6. The **service** procedure gets the first message (*q_first*) from the message queue with the **getq(9)** utility.
7. The **service** procedure processes the message and passes it to the **put** procedure of the next queue with **putnext(9)**.
8. The **service** procedure gets the next message and processes it.

This processing continues until the queue is empty or flow control blocks further processing. The **service** procedure returns to the caller.

A **service** procedure must never sleep since it has no user context. It must always return to its caller.

If no processing is required in the **put** procedure, the procedure does not have to be explicitly declared. Rather, **putq(9)** can be placed in the **qinit** structure declaration for the appropriate queue side to queue the message for the **service** procedure, e.g.,

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

More typically, **put** procedures will, at a minimum, process high priority messages to avoid queueing them.

The key attribute of a **service** procedure in the *STREAMS* architecture is delayed processing. When a **service** procedure is used in a module, the module developer is implying that there are other, more time sensitive activities to be performed elsewhere in this *Stream*, in other *Streams*, or in the system in general. The presence of a **service** procedure is mandatory if the flow control mechanism is to be utilized by the queue.

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

If a module or driver wishes to recognize priority bands, the **service** procedure is written to the following algorithm:

```
.
.
while ((bp = getq(q)) != NULL) {
    if (bp->b_datap->db_type >= QPCTL) {
        putnext(q, bp);
    } else if (bcanput(q, bp->b_band)) {
        putnext(q, bp);
    } else {
        putbq(q, bp);
    }
}
```

```

        return;
    }
}
.
.

```

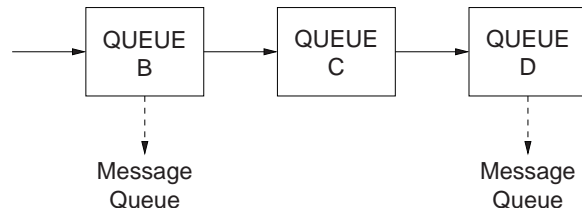
5.3.2.1 Flow Control

The *STREAMS* flow control mechanism is voluntary and operates between the two nearest queues in a *Stream* containing **service** procedures (see [Figure 5.5](#)). Messages are generally held on a queue only if a **service** procedure is present in the associated queue.

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

Flow control operates as follows:

1. Each time a *STREAMS* message handling routine (for example, `putq(9)`) adds or removes a message from a message queue, the limits are checked. *STREAMS* calculates the total size of all message blocks (`'bp->b_wptr - bp->b_rptr'`) on the message queue.
2. The total is compared to the queue high water and low water values. If the total exceeds the high water value, an internal full indicator is set for the queue. The operation of the **service** procedure in this queue is not affected if the indicator is set, and the **service** procedure continues to be scheduled.
3. The next part of flow control processing occurs in the nearest preceding queue that contains a **service** procedure. In [Figure 5.5](#), if 'D' is full and 'C' has no **service** procedure, then 'B' is the nearest preceding queue.
4. The service procedure in 'B' uses a *STREAMS* utility routine to see if a queue ahead is marked full. If messages cannot be sent, the scheduler blocks the service procedure in 'B' from further execution. 'B' remains blocked until the low water mark of the full queue, 'D', is reached.
5. While 'B' is blocked, any messages except high priority messages arriving at 'B' will accumulate on its message queue (recall that high priority messages are not subject to flow control). Eventually, 'B' may reach a full state and the full condition will propagate back to the module in the *Stream*.
6. When the **service** procedure processing on 'D' causes the message block total to fall below the low water mark, the full indicator is turned off. Then, *STREAMS* automatically schedules the nearest preceding blocked queue ('B' in this case), getting things moving again. This automatic scheduling is known as back-enabling a queue.

Figure 5.5: *Flow Control*

Modules and drivers need to observe the message priority. High priority messages, determined by the type of the first block in the message, (`'mp->b_datap->db_type >= QPCTL'`), are not subject to flow control. They are processed immediately and forwarded, as appropriate.

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

This is the general flow control processing of ordinary messages:

1. Retrieve the message at the head of the queue with `getq(9)`.
2. Determine if the message type is high priority and not to be processed here.
3. If so, pass the message to the `put` procedure of the following queue with `putnext(9)`.
4. Use `canput(9)` to determine if messages can be sent onward.
5. If messages should not be forwarded, put the message back on the queue with `putbq(9)` and return from the procedure.
6. Otherwise, process the message.

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

```

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

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

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

The call `'bcanput(q, 0);'` is equivalent to the call `'canput(q);'`.

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

A **service** procedure continues processing messages from its queue until **getq(9)** returns 'NULL'. When an ordinary message is enqueued by **putq(9)**, **putq()** will cause the **service** procedure to be scheduled only if the queue was previously empty, and a previous **getq()** call returns 'NULL' (that is, the **QWANTR** flag is set). If there are messages on the queue, **putq()** presumes the **service** procedure is blocked by flow control and the procedure will be automatically rescheduled by *STREAMS* when the block is removed. If the **service** procedure cannot complete processing as a result of conditions other than flow control (e.g., no buffers), it must ensure it will return later [e.g., by use of **bufcall(9)** utility routine] or it must discard all messages on the queue. If this is not done, *STREAMS* will never schedule the **service** procedure to be run unless the queue's **put** procedure enqueues a priority message with **putq()**.

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

putbq(9) replaces messages at the beginning of the appropriate section of the message queue in accordance with their priority. This might not be the same position at which the message was retrieved by the preceding **getq(9)**. A subsequent **getq()** might return a different message.

putq(9) only looks at the priority band in the first message. If a high priority message is passed to **putq()** with a non-zero *b_band* value, *b_band* is reset to 0 before placing the message on the queue.¹ If the message is passed to **putq()** with a *b_band* value that is greater than the number of **qband** structures associated with the queue, **putq()** tries to allocate a new **qband** structure for each band up to and including the band of the message.²

The above also applies to **putbq(9)** and **insq(9)**. If an attempt is made to insert a message out of order in a queue via **insq()**, the message is not inserted and the routine fails.

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

Driver upstream flow control is explained next as an example. Although device drivers typically discard input when unable to send it to a user process, *STREAMS* allows driver read-side flow control, possibly for handling temporary upstream blockages. This is done through a driver read **service** procedure which is disabled during the driver open with **noenable(9)**. If the driver input interrupt routine determines messages can be sent upstream (from **canput(9)**), it sends the message with **putnext(9)**. Otherwise, it calls **putq(9)** to queue the message. The message waits on the message queue (possibly with queue length checked when new messages are enqueued by the interrupt routine) until the upstream queue becomes unblocked. When the blockage abates, *STREAMS* back-enables

¹ Check that Linux Fast-STREAMS actually performs this action.

² Linux Fast-STREAMS currently only allocated the band in question and does not allocate intermediate queue band structures as described.

the driver read `service` procedure. The `service` procedure sends the messages upstream using `getq(9)` and `canput(9)`, as described previously. This is similar to `looprsrv()` (see [Section 9.3 \[Loop-Around Driver\]](#), page 161) where the `service` procedure is present only for flow control.

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

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

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

5.4 STREAMS Service Interfaces

STREAMS provides the means to implement a service interface between any two components in a *Stream*, and between a user process and the topmost module in the *Stream*. A service interface is defined at the boundary between a service user and a service provider (see [Figure 5.7](#)). A service interface is a set of primitives and the rules that define a service and the allowable state transitions that result as these primitives are passed between the user and the provider. These rules are typically represented by a state machine. In *STREAMS*, the service user and provider are implemented in a module, driver, or user process. The primitives are carried bidirectionally between a service user and provider in `M_PROTO` and `M_PCPROTO` messages.

PROTO messages (`M_PROTO` and `M_PCPROTO`) can be multi-block, with the second through last blocks of type `M_DATA`. The first block in a *PROTO* message contains the control part of the primitive in a form agreed upon by the user and provider. The block is not intended to carry protocol headers. (Although its use is not recommended, upstream *PROTO* messages can have multiple *PROTO* blocks at the start of the message. `getmsg(2)` will compact the blocks into a single control part when sending to a user process.) The `M_DATA` block(s) contains any data part associated with the primitive. The data part may be processed in a module that receives it, or it may be sent to the next *Stream* component, along with any data generated by the module. The contents of *PROTO* messages and their allowable sequences are determined by the service interface specification.

PROTO messages can be sent bidirectionally (upstream and downstream) on a *Stream* and between a *Stream* and a user process. `putmsg(2)` and `getmsg(2)` system calls are analogous, respectively, to `write(2)` and `read(2)` except that the former allow both data and control

parts to be (separately) passed, and they retain the message boundaries across the user-*Stream* interface. `putmsg(2)` and `getmsg(2)` separately copy the control part (`M_PROTO` or `M_PCPROTO` block) and data part (`M_DATA` blocks) between the *Stream* and user process.

An `M_PCPROTO` message is normally used to acknowledge primitives composed of other messages. `M_PCPROTO` insures that the acknowledgement reaches the service user before any other message. If the service user is a user process, the *Stream head* will only store a single `M_PCPROTO` message, and discard subsequent `M_PCPROTO` messages until the first one is read with `getmsg(2)`.

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

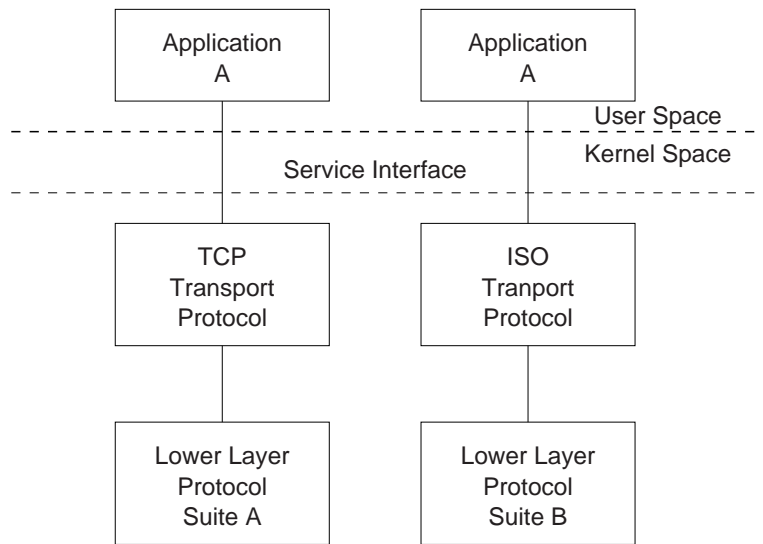
This section describes the system calls `getmsg` and `putmsg` in the context of a service interface example. First, a brief overview of *STREAMS* service interfaces is presented.

5.4.1 Service Interface Benefits

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

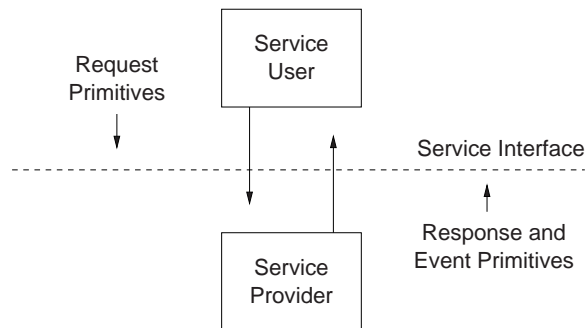
One benefit of modularity is the ability to interchange modules of like functions. For example, two distinct transport protocols, implemented as *STREAMS* modules, may provide a common set of services. An application or higher layer protocol that requires those services can use either module. This ability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

Each *STREAMS* module provides a set of processing functions, or services, and an interface to those services. The service interface of a module defines the interaction between that module and any neighboring modules, and is a necessary component for providing module substitution. By creating a well defined service interface, applications and *STREAMS* modules can interact with any module that supports that interface. [Figure 5.6](#) demonstrates this.

Figure 5.6: *Protocol Substitution*

By defining a service interface through which applications interact with a transport protocol, it is possible to substitute a different protocol below that service interface in a manner completely transparent to the application. In this example, the same application can run over the *Transmission Control Protocol (TCP)* and the *ISO* transport protocol. Of course, the service interface must define a set of services common to both protocols.

The three components of any service interface are the service user, the service provider, and the service interface itself, as seen in the following figure.

Figure 5.7: *Service Interface*

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

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

An example of a service interface primitive is a transport protocol connect request. This primitive requests the transport protocol service provider to establish a connection with another transport user. The parameters associated with this primitive may include a destination protocol address and specific protocol options to be associated with that connection. Some transport protocols also allow a user to send data with the connect request. A *STREAMS* message would be used to define this primitive. The control part would identify the primitive as a connect request and would include the protocol address and options. The data part would contain the associated user data.

5.4.2 Service Interface Library Example

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

- establish a *Stream* to the service provider and bind a protocol address to the *Stream*,
- send data to a remote user,
- close the *Stream* connected to the provider

First, the structure and constant definitions required by the library are shown. These typically will reside in a header file associated with the service interface.

```

/*
 * Primitives initiated by the service user.
 */
#define BIND_REQ 1          /* bind request */
#define UNITDATA_REQ 2     /* unitdata request */

/*
 * Primitives initiated by the service provider.
 */
#define OK_ACK 3           /* bind acknowledgment */
#define ERROR_ACK 4       /* error acknowledgment */
#define UNITDATA_IND 5    /* unitdata indication */

/*
 * The following structure definitions define the format of the
 * control part of the service interface message of the above
 * primitives.
 */
struct bind_req {          /* bind request */
    long PRIM_type;       /* always BIND_REQ */
    long BIND_addr;       /* addr to bind */
};
struct unitdata_req {     /* unitdata request */
    long PRIM_type;       /* always UNITDATA_REQ */
    long BEST_addr;       /* destination addr */
};
struct ok_ack {           /* positive acknowledgment */
    long PRIM_type;       /* always OK_ACK */
};

```



```

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

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

/* header files needed by library */
#include <stropts.h>
#include <stdio.h>
#include <errno.h>

```

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

BIND_REQ

This request asks the provider to bind a specified protocol address. It requires an acknowledgement from the provider to verify that the contents of the request were syntactically correct.

UNITDATA_REQ

This request asks the provider to send data to the specified destination address. It does not require an acknowledgement from the provider.

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

OK_ACK This primitive informs the user that a previous bind request was received successfully by the service provider.

ERROR_ACK

This primitive informs the user that a non-fatal error was found in the previous bind request. It indicates that no action was taken with the primitive that caused the error.

UNITDATA_IND

This primitive indicates that data destined for the user has arrived.

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

5.4.2.1 Accessing the Service Provider

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

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

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

    /* send bind request msg down stream */

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

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

After opening the protocol driver, `inter_open` packages a bind request message to send downstream. `putmsg(2)` is called to send the request to the service provider. The bind request message contains a control part that holds a `bind_req` structure, but it has no data part. `bind_req` is a structure of type `strbuf`, and it is initialized with the primitive type and address. Notice that the `maxlen` field of `ctlbuf` is not set before calling `putmsg(2)`. That is because `putmsg(2)` ignores this field. The `dataptr` argument to `putmsg(2)` is set to ‘NULL’ to indicate that the message contains no data part. Also, the `flags` argument is ‘0’, which specifies that the message is not a high priority message.

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

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

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

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

```

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

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

case OK_ACK:
    return (fd);

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

```

`getmsg(2)` is called to retrieve the acknowledgement of the bind request. The acknowledgement message consists of a control part that contains either an `ok_ack` or `error_ack` structure, and no data part.

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

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

Because neither acknowledgement primitive contains a data part, the `dataptr` argument to `getmsg(2)` is set to 'NULL'. The `flagsp` argument points to an integer containing the value `RS_HIPRI`. This flag indicates that `getmsg(2)` should wait for a *STREAMS* high priority message before returning. It is set because we want to catch the acknowledgement primitives that are priority messages. Otherwise if the flag is zero the first message is taken.

With `RS_HIPRI` set, even if a normal message is available, `getmsg(2)` will block until a high priority message arrives.

On return from `getmsg(2)`, the `len` field is checked to ensure that the control part of the retrieved message is an appropriate size. The example then checks the primitive type and takes appropriate actions. An `OK_ACK` indicates a successful bind operation, and `inter_open` returns the file descriptor of the open *Stream*. An `[ERROR_ACK]` indicates a bind failure, and `errno(3)` is set to identify the problem with the request.

5.4.2.2 Closing the Service Provider

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

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

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

5.4.2.3 Sending Data to the Service Provider

The third routine, `inter_snd`, passes data to the service provider for transmission to the user at the address specified in `addr`. The data to be transmitted are contained in the buffer pointed to by `buf` and contains `len` bytes. On successful completion, this routine returns the number of bytes of data passed to the service provider; on failure, it returns ‘-1’ and sets `errno(3)` to an appropriate *UNIX* system error value.

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

    unitdata_req.PRIM_type = UNITDATA_REQ;
    unitdata_req.DEST_addr = addr;
    ctlbuf.len = sizeof(struct unitdata_req);
    ctlbuf.buf = (char *) &unitdata_req;
    databuf.len = len;
    databuf.buf = buf;

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

    return (len);
}
```

In this example, the data request primitive is packaged with both a control part and a data part. The control part contains a `unitdata_req` structure that identifies the primitive type

and the destination address of the data. The data to be transmitted are placed in the data part of the request message.

Unlike the bind request, the data request primitive requires no acknowledgement from the service provider. In the example, this choice was made to minimize the overhead during data transfer. If the `putmsg(2)` call succeeds, this routine assumes all is well and returns the number of bytes passed to the service provider.

5.4.2.4 Receiving Data

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

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

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

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

`getmsg(2)` is called to retrieve the data indication primitive, where that primitive contains both a control and data part. The control part consists of a `unitdata_ind` structure that identifies the primitive type and the source address of the data sender. The data part contains the data itself.

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

The integer pointed at by *flagsp* in the `getmsg(2)` call is set to zero, indicating that the next message should be retrieved from the *Stream head*, regardless of its priority. Data will arrive in normal priority messages. If no message currently exists at the *Stream head*, `getmsg` will block until a message arrives.

The user's control and data buffers should be large enough to hold any incoming data. If both buffers are large enough, `getmsg(2)` will process the data indication and return '0', indicating that a full message was retrieved successfully. However, if either buffer is not large enough, `getmsg(2)` will only retrieve the part of the message that fits into each user buffer. The remainder of the message is saved for subsequent retrieval (if in message non-discard mode), and a positive, non-zero value is returned to the user. A return value of 'MORECTL' indicates that more control information is waiting for retrieval. A return value of 'MOREDATA' indicates that more data are waiting for retrieval. A return value of '(MORECTL | MOREDATA)' indicates that data from both parts of the message remain. In the example, if the user buffers are not large enough (that is, `getmsg(2)` returns a positive, non-zero value), the function will set [EIO] to `errno(3)` and fail.

The type of the primitive returned by `getmsg(2)` is checked to make sure it is a data indication (`UNITDATA_IND` in the example). The source address is then set and the number of bytes of data is returned.

The example presented is a simplified service interface. The state transition rules for such an interface were not presented for the sake of brevity. The intent was to show typical uses of the `putmsg(2)` and `getmsg(2)` system calls. See `putmsg(2)` and `getmsg(2)` for further details. For simplicity, this example did not also consider expedited data.

5.4.2.5 Module Service Interface Example

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

- Modules and drivers that support a service interface must act upon all *PROTO* messages and not pass them through.
- Modules may be inserted between a service user and a service provider to manipulate the data part as it passes between them. However, these modules may not alter the contents of the control part (*PROTO* block, first message block) nor alter the boundaries of the control or data parts. That is, the message blocks comprising the data part may be changed, but the message may not be split into separate messages nor combined with other messages.

In addition, modules and drivers must observe the rule that high priority messages are not subject to flow control and forward them accordingly.

Declarations

The service interface primitives are defined in the declarations:

```

#include <sys/types.h>
#include <sys/param.h>
#include <sys/stream.h>
#include <sys/errno.h>

/* Primitives initiated by the service user */

#define BIND_req 1          /* bind request */
#define UNITDATA_REQ 2      /* unitdata request */

/* Primitives initiated by the service provider */

#define OK_ACK 3            /* bind acknowledgment */
#define ERROR_ACK 4        /* error acknowledgment */
#define UNITDATA_IND 5     /* unitdata indication */
/*
 * The following structures define the format of the
 * stream message block of the above primitives.
 */
struct bind_req {          /* bind request */
    long PRIM_type;        /* always BIND_REQ */
    long BIND_addr;        /* addr to bind */
};
struct unitdata_req {      /* unitdata request */
    long PRIM_type;        /* always UNITDATA_REQ */
    long DEST_addr;        /* dest addr */
};
struct ok_ack {            /* ok acknowledgment */
    long PRIM_type;        /* always OK_ACK */
};
struct error_ack {         /* error acknowledgment */
    long PRIM_type;        /* always ERROR_ACK */
    long UNIX_error;       /* UNIX system error code */
};
struct unitdata_ind {      /* unitdata indication */
    long PRIM_type;        /* always UNITDATA_IND */
    long SRC_addr;         /* source addr */
};
union primitives {         /* union of all primitives */
    long type;
    struct bind_req bind_req;
    struct unitdata_req unitdata_req;
    struct ok_ack ok_ack;
    struct error_ack error_ack;
    struct unitdata_ind unitdata_ind;
};
struct dgproto {           /* structure per minor device */
    short state;           /* current provider state */
    long addr;             /* net address */
};

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

```

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

Two commands are recognized by the module:

BIND_REQ

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

UNITDATA_REQ

Send data to the specified address.

Three messages are generated:

OK_ACK A positive acknowledgement (ack) of `BIND_REQ`.

ERROR_ACK

A negative acknowledgement (nak) of `BIND_REQ`.

UNITDATA_IND

Data from the network have been received (this code is not shown).

The acknowledgement of a `BIND_REQ` informs the user that the request was syntactically correct (or incorrect if `[ERROR_ACK]`). The receipt of a `BIND_REQ` is acknowledged with an `M_PCPROTO` to insure that the acknowledgement reaches the user before any other message. For example, a `UNITDATA_IND` could come through before the bind has completed, and the user would get confused.

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

state current state of the service provider `IDLE` or `BOUND`
addr network address that has been bound to this *Stream*

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

Service Interface Procedure

The write put procedure is:

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

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

    switch (mp->b_datap->db_type) {

    default:
        /* don't understand it */
```



```

    mp->b_datap->db_type = M_ERROR;
    mp->b_rptr = mp->b_wptr = mp->b_datap->db_base;
    *mp->b_wptr++ = EPROTO;
    qreply(q, mp);
    break;

case M_FLUSH:
    /* standard flush handling goes here ... */
    break;

case M_PROTO:
    /* Protocol message -> user request */
    proto = (union primitives *) mp->b_rptr;

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

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

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

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

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

```

```

        goto bad;
        putq(q, mp);
        /* start device or mux output ... */
    }
    break;

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

```

The write `put` procedure switches on the message type. The only types accepted are `M_FLUSH` and `M_PROTO`. For `M_FLUSH` messages, the driver will perform the canonical flush handling (not shown). For `M_PROTO` messages, the driver assumes the message block contains a union primitive and switches on the type field. Two types are understood: `BIND_REQ` and `UNITDATA_REQ`.

For a `BIND_REQ`, the current state is checked; it must be `IDLE`. Next, the message size is checked. If it is the correct size, the passed-in address is verified for legality by calling `chkaddr`. If everything checks, the incoming message is converted into an `OK_ACK` and sent upstream. If there was any error, the incoming message is converted into an `[ERROR_ACK]` and sent upstream.

For `UNITDATA_REQ`, the state is also checked; it must be `BOUND`. As above, the message size and destination address are checked. If there is any error, the message is simply discarded. If all is well, the message is put on the queue, and the lower half of the driver is started.

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

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

5.5 STREAMS Message Allocation and Freeing

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

`allocb()` may return a buffer larger than the size requested. If `allocb()` indicates buffers are not available [`allocb()` fails], the `put/service` procedure may not call `sleep(9)` to wait for a buffer to become available. Instead, the `bufcall(9)` utility can be used to defer processing in the module or the driver until a buffer becomes available.

If message space allocation is done by the `put` procedure and `allocb()` fails, the message is usually discarded. If the allocation fails in the `service` routine, the message is returned to the queue. `bufcall(9)` is called to enable to the `service` routine when a message buffer becomes available, and the `service` routine returns.

The `freeb(9)` utility routine releases (de-allocates) the message block descriptor and the corresponding data block, if the reference count (see `datab` structure) is equal to '1'. If the reference counter exceeds '1', the data block is not released.

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

In the following example, `allocb()` is used by the `bappend` subroutine that appends a character to a message block:

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

#define MODBLKSZ 128                /* size of message blocks */

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

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

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

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

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

```
/* Write side put procedure */
static
modwput(q, mp)
    queue_t *q;
    mblk_t *mp;
```

```

{
    switch (mp->b_datap->db_TYPE) {
    default:
        putnext(q, mp);          /* Don't do these, pass them along */
        break;

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

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

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

```

Data messages are scanned and filtered. `modwput1` copies the original message into a new block(s), modifying as it copies. *nbp* points to the current new message block. *nmp* points to the new message being formed as multiple `M_DATA` message blocks. The outer `for()` loop goes through each message block of the original message. The inner `while()` loop goes through each byte. `bappend` is used to add characters to the current or new block. If `bappend` fails, the current new block is full. If *nmp* is 'NULL', *nmp* is pointed at the new block. If *nmp* is not 'NULL', the new block is linked to the end of *nmp* by use of the `linkb(9)` utility.

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

5.5.0.1 Recovering From No Buffers

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

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

`bufcall()` calls ‘`(*func)(arg)`’ when a buffer of size bytes is available. When *func* is called, it has no user context and must return without sleeping. Also, because of interrupt processing, there is no guarantee that when *func* is called, a buffer will actually be available (someone else may steal it).

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

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

Two examples are provided. The first example is a device receive interrupt handler:

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

dev_rintr(dev)
{
    /* process incoming message ... */

    /* allocate new buffer for device */
    dev_re_load(dev);
}

/*
 * Reload device with a new receive buffer
 */
dev_re_load(dev)
{
    mblk_t *bp;

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

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

`dev_rintr` is called when the device has posted a receive interrupt. The code retrieves the data from the device (not shown). `dev_rintr` must then give the device another buffer to fill by a call to `dev_re_load`, which calls `allocb(9)`. If `allocb()` fails, `dev_re_load` uses `bufcall(9)` to call itself when *STREAMS* determines a buffer is available.

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

The second example is a write **service** procedure, `mod_wsrv`, which needs to prepend each output message with a header. `mod_wsrv` illustrates a case for potential deadlock:

```
static int
mod_wsrv(q)
    queue_t *q;
{
    int qenable();
    mblk_t *mp, *bp;

    while (mp = getq(q)) {
        /* check for priority messages and canput ... */

        /* Allocate a header to prepend to the message. If the allocb
           fails, use bufcall to reschedule. */
        if ((bp = allocb(HDRSZ, BPRI_MED)) == NULL) {
            if (!bufcall(HDRSZ, BPRI_MED, qenable, q)) {
                timeout(qenable, q, HZ * 2);
            }
            /* Put the message back and exit, we will be re-enabled
               later */
            putbq(q, mp);
            return;
        }
        /* process message ... */
    }
}
```

However, if `allocb(9)` fails, `mod_wsrv` wants to recover without loss of data and calls `bufcall(9)`. In this case, the routine passed to `bufcall()` is `qenable(9)`. When a buffer is available, the **service** procedure will be automatically re-enabled. Before exiting, the current message is put back on the queue. This example deals with `bufcall()` failure by resorting to the `timeout(9)` operating system utility routine. `timeout()` will schedule the given function to be run with the given argument in the given number of clock ticks (there are HZ ticks per second). In this example, if `bufcall()` fails, the system will run `qenable(9)` after two seconds have passed.³

5.6 STREAMS Extended Buffers

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

If the hardware supports memory-mapped I/O, data received from the network are placed in the *DARAM* (*dual access RAM*) section of the I/O card. Since *DARAM* is a shared memory between the kernel and the I/O card, data transfer between the kernel and the I/O

³ Under Linux Fast-STREAMS, it is not a good strategy to call `timeout(9)` when `bufcall(9)` fails. This is because if `bufcall(9)` fails to allocate resource for the buffer callback, `timeout(9)` is also likely to fail to allocate necessary resources. As a final resort, if `timeout(9)` fails, the **service** procedure can simply re-enable itself with `qenable(9)` and try again when it is rescheduled.

card is eliminated. Once in kernel space, the data buffer can be manipulated as if it were a kernel resident buffer. Similarly, data being sent downstream are placed in *DARAM* and then forwarded to the network.

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

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

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

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

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

The format of the `free_rtn` structure is as follows:

```
struct free_rtn {
    void (*free_func) ();          /* driver dependent free routine */
    char *free_arg;               /* argument for free_rtn */
};
typedef struct free_rtn frtn_t;
```

The structure has two fields: a pointer to a function and a location for any argument passed to the function. Instead of defining a specific number of arguments, `free_arg` is defined as

a `'char *'`. This way, drivers can pass pointers to structures in the event more than one argument is needed.

The *STREAMS* utility routine, `esballoc(9)`, provides a common interface for allocating and initializing data blocks. It makes the allocation as transparent to the driver as possible and provides a way to modify the fields of the data block, since modification should only be performed by *STREAMS*. The driver calls this routine when it wants to attach its own data buffer to a newly allocated message and data block. If the routine successfully completes the allocation and assigns the buffer, it returns a pointer to the message block. The driver is responsible for supplying the arguments to `esballoc(9)`, namely, a pointer to its data buffer, the size of the buffer, the priority of the data block, and a pointer to the `free_rtn` structure. All arguments should be non-`NULL`. See [Appendix C \[STREAMS Utilities\]](#), [page 295](#), for a detailed description of `esballoc(9)`.

6 Polling and Signalling

6.1 STREAMS Input and Output Polling

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

User processes can efficiently monitor and control multiple *Streams* with two system calls: `poll(2)` and the `I_SETSIG ioctl(2)` command. These calls allow a user process to detect events that occur at the *Stream head* on one or more *Streams*, including receipt of data or messages on the read queue and cessation of flow control.

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

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

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

6.1.1 Synchronous Input and Output

The `poll(2)` system call provides a mechanism to identify those *Streams* over which a user can send or receive data. For each *Stream* of interest users can specify one or more events about which they should be notified. The types of events that can be polled are `POLLIN`, `POLLRDNORM`, `POLLRDBAND`, `POLLPRI`, `POLLOUT`, `POLLWRNORM`, `POLLWRBAND`, `POLLMSG`, and `POLLNORM`:

POLLIN A message other than an `M_PCPROTO` is at the front of the *Stream head* read queue. This event is maintained for compatibility with the previous releases of the *UNIX System V*.

POLLRDNORM A normal (non-priority) message is at the front of the *Stream head* read queue.

POLLRDBAND A priority message (`'band > 0'`) is at the front of the *Stream head* read queue.

POLLPRI A high priority message (`M_PCPROTO`) is at the front of the *Stream head* read queue.

POLLOUT The normal priority band of the queue is writable (not flow controlled).

POLLWRNORM
The same as POLLOUT.

POLLWRBAND
A priority band greater than 'O' of a queue downstream exists and is writable.

POLLMSG An `M_SIG` or `M_PCSIG` message containing the `{SIGPOLL}` signal has reached the front of the *Stream head* read queue.

POLLNORM The same as `POLLRDNORM`.

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

The `poll` system call will examine each file descriptor for the requested events and, on return, will indicate which events have occurred for each file descriptor. If no event has occurred on any polled file descriptor, `poll` blocks until a requested event or timeout occurs. `poll(2)` takes the following arguments:

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

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

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

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

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

    if ((pollfds[0].fd = open("/dev/comm/01", O_RDWR | O_NDELAY)) < 0) {
        perror("open failed for /dev/comm/01");
        exit(1);
    }
    if ((pollfds[1].fd = open("/dev/comm/02", O_RDWR | O_NDELAY)) < 0) {
        perror("open failed for /dev/comm/02");
```

```
    exit(2);
}
```

The variable `pollfds` is declared as an array of the `pollfd` structure that is defined in ‘`poll.h`’ and has the following format:

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

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

The example continues to process incoming data as follows:

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

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

            default:          /* default error case */
                perror("error event");
                exit(4);

            case 0:           /* no events */
                break;

            case POLLIN:
                /* echo incoming data on "other" Stream */
                while ((count = read(pollfds[i].fd, buf, 1024)) > 0)
                    /* the write loses data if flow control prevents the
                     * transmit at this time. */
                    if (write(pollfds[(i + 1) % 2].fd, buf, count) != count)
                        fprintf(stderr, "writer lost data \n");
                break;
        }
    }
}
```

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

The second argument to the `poll` system call specifies the number of entries in the *pollfds* array (‘2’ in this example). The third argument is a timeout value indicating the number of milliseconds `poll` should wait for an event if none has occurred. On a system where millisecond accuracy is not available, timeout is rounded up to the nearest value available

on that system. If the value of timeout is 0, `poll` returns immediately. Here, the value of timeout is '-1', specifying that `poll` should block until a requested event occurs or until the call is interrupted.

If the `poll` call succeeds, the program looks at each entry in the `pollfds` array. If `revents` is set to '0', no event has occurred on that file descriptor. If `revents` is set to `POLLIN`, incoming data are available. In this case, all available data are read from the polled minor device and written to the other minor device.

If `revents` is set to a value other than '0' or `POLLIN`, an error event must have occurred on that *Stream*, because `POLLIN` was the only requested event. The following are `poll` error events:

- POLLERR** A fatal error has occurred in some module or driver on the *Stream* associated with the specified file descriptor. Further system calls will fail.
- POLLHUP** A hangup condition exists on the *Stream* associated with the specified file descriptor. This event and `POLLOUT` are mutually exclusive; a *Stream* can't be writable if a hangup has occurred.
- POLLNVAL** The specified file descriptor is not associated with an open *Stream*.

These events may not be polled for by the user, but will be reported in `revents` whenever they occur. As such, they are only valid in the `revents` bitmask.

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

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

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

6.1.2 Asynchronous Input and Output

The `poll` system call described before enables a user to monitor multiple *Streams* in a synchronous fashion. The `poll(2)` call normally blocks until an event occurs on any of the polled file descriptors. In some applications, however, it is desirable to process incoming data asynchronously. For example, an application may wish to do some local processing and be interrupted when a pending event occurs. Some time-critical applications cannot afford to block, but must have immediate indication of success or failure.

The `I_SETSIG` `ioctl(2)` call [see `streamio(7)`] is used to request that a `{SIGPOLL}` signal be sent to a user process when a specific event occurs. Listed below are events for the `ioctl` `I_SETSIG`. These are similar to those described for `poll(2)`.

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

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

6.1.3 Signals

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

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

6.1.3.1 Extended Signals

To enable a process to obtain the band and event associated with `{SIGPOLL}` more readily, *STREAMS* supports extended signals. For the given events, a special code is defined in ‘`siginfo.h`’ that describes the reason `{SIGPOLL}` was generated. The following table describes the data available in the `siginfo_t` structure passed to the signal handler.

event	si_signo	si_code	si_band	si_errno
S_INPUT	{SIGPOLL}	POLL_IN	band readable	unused
S_OUTPUT	{SIGPOLL}	POLL_OUT	band writable	unused
S_MSG	{SIGPOLL}	POLL_MSG	band signaled	unused
S_ERROR	{SIGPOLL}	POLL_ERR	unused	<i>Stream</i> error
S_HANGUP	{SIGPOLL}	POLL_HUP	unused	unused
S_HIPRI	{SIGPOLL}	POLL_PRI	unused	unused

6.2 STREAMS Stream as Controlling Terminal

6.2.1 Job Control

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

Job Control is a feature supported by the *BSD UNIX* operating system. It is also an optional part of the *IEEE P1003.1 POSIX* standard. *Job Control* breaks a login session into smaller units called jobs. Each job consists of one or more related and cooperating processes. One job, the foreground job, is given complete access to the controlling terminal. The other jobs, background jobs, are denied read access to the controlling terminal and given conditional write and `ioctl` access to it. The user may stop an executing job and resume the stopped job either in the foreground or in the background.

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

The following defines terms associated with *Job Control*:

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

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

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

{SIGCONT}

Sent to a stopped process to continue it.

{SIGSTOP}

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

{SIGTSTP}

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

{SIGTTIN}

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

{SIGTTOU}

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

A session may be allocated a controlling terminal. For every allocated controlling terminal, *Job Control* elevates one process group in the controlling process's session to the status of foreground process group. The remaining process groups in the controlling process's session are background process groups. A controlling terminal gives a user the ability to control execution of jobs within the session. Controlling terminals play a central role in *Job Control*. A user may cause the foreground job to stop by typing a predefined key on the controlling terminal. A user may inhibit access to the controlling terminal by background jobs. Background jobs that attempt to access a terminal that has been so restricted will be sent a signal that typically will cause the job to stop. (see [\[Hangup Signals\]](#), page 103, later in this chapter.)

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

SUSP character

A user defined character that, when typed, causes the line discipline module to request that the *Stream* head sends a {SIGTSTP} signal to the foreground process with an M_PCSIG message, which by default stops the members of that group. If the value of SUSP is zero, the {SIGTSTP} signal is not sent, and the SUSP character is disabled.

TOSTOP flag

If TOSTOP is set, background processes are inhibited from writing to their controlling terminal.

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

6.2.2 Allocation and Deallocation

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

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

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

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

6.2.3 Hung-up Streams

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

If the reopen is successful, the hung-up condition is cleared.

6.2.4 Hangup Signals

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

6.2.5 Accessing the Controlling Terminal

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

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

- If the background process is ignoring or holding the `{SIGTTIN}` signal or is a member of an orphaned process group, an attempt to read from the controlling terminal will fail with an `[EIO]` error. Otherwise, the process is sent a `{SIGTTIN}` signal, which by default stops the process.
- If the process is attempting to write to the terminal and if the terminal's `TOSTOP` flag is clear, the process is allowed access. The `TOSTOP` flag is set upon reception of an `M_SETOPTS` message with the `SO_TOSTOP` flag set in the `so_flags` field. It is cleared upon reception of an `M_SETOPTS` message with the `SO_TONSTOP` flag set.
- If the terminal's `TOSTOP` flag is set and a background process is attempting to write to the terminal, the write will succeed if the process is ignoring or holding `{SIGTTOU}`. Otherwise, the process will stop except when it is a member of an orphaned process group, in which case it is denied access to the terminal and it is returned an `[EIO]` error.
- If a background process is attempting to perform a destructive `ioctl` (an `ioctl` that modifies terminal parameters), the `ioctl` call will succeed if the process is ignoring or holding `{SIGTTOU}`. Otherwise, the process will stop except when the process is a member of the orphaned process group. In that case the access to the terminal is denied and an `[EIO]` error is returned.

7 Overview of STREAMS Modules and Drivers

7.1 STREAMS Module and Driver Environment

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

STREAMS modules and drivers are structurally similar. The call interfaces to driver routines are identical to interfaces used for modules. Drivers and modules must declare `streamtab(9)`, `qinit`, and `module_info` structures. Within the *STREAMS* mechanism drivers are required elements, but modules are optional. However, in the *STREAMS*-based pipe mechanism and the pseudo-terminal subsystem only the *Stream* head is required.

There are three significant differences between modules and drivers. A driver must be able to handle interrupts from a device, so the driver will typically include an interrupt handler routine. Another difference is that a driver may have multiple *Streams* connected to it. The third difference is the initialization/deinitialization process that happens via `open/close` with a driver and via the `ioctl`s `I_PUSH/I_POP` with a module. (`I_PUSH/I_POP` results in calls to `open/close`.)

User context is not generally available to *STREAMS* module procedures and drivers. The exception is during execution of the `open` and `close` routines. Driver and module `open` and `close` routines have user context and may access the `current task_struct` structure, although this is discouraged. These routines are allowed to sleep, but must always return to the caller. That is, if they sleep, it must be at priority numerically \leq `PZERO`, or with `PCATCH` set in the sleep priority. Priorities are higher as they decrease in numerical value. The process will never return from the sleep call and the system call will be aborted if:

- A process is sleeping at priority $> PZERO$,
- `PCATCH` is not set, and
- A process is sent signal via `kill(2)`.

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

The module and driver `open/close` interface has been modified for *UNIX System V Release 4.0*. However, the system defaults to *UNIX System V Release 3.0* interface unless `prefixflag` is defined. This is discussed later in this chapter (see [Section 7.4 \[STREAMS Driver-Kernel Interface\]](#), page 128). Examples and descriptions in this chapter reflect *Release 4.0* interface.

7.1.1 Module and Driver Declarations

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

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

static struct module_info rminfo = { 0x08, "mod", 0, INFPSZ, 0, 0 };
static struct module_info wminfo = { 0x08, "mod", 0, INFPSZ, 0, 0 };
```

```

static int modopen(), modput(), modclose();

static struct qinit rinit = {
    modput, NULL, modopen, modclose, NULL, &rminfo, NULL
};

static struct qinit winit = {
    modput, NULL, NULL, NULL, NULL, &wminfo, NULL
};

struct streamtab modinfo = { &rinit, &winit, NULL, NULL };

extern int moddevflag = 0;

```

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

The declarations shown are: the header set; the read and write queue (**rminfo** and **wminfo**) **module_info** structures; the module open, read-put, write-put, and close procedures; the read and write (**rinit**, and **winit**) **qinit** structures; and the **streamtab(9)** structure.

The header files, ‘types.h’ and ‘stream.h’, are always required for modules and drivers. The header file, ‘param.h’, contains definitions for ‘NULL’ and other values for *STREAMS* modules and drivers as shown later in this chapter (see [Section 7.5.4 \[Accessible Symbols and Functions\]](#), page 135).

When configuring a *STREAMS* module or driver (see [Appendix E \[STREAMS Configuration\]](#), page 323) the **streamtab** structure must be externally accessible. The **streamtab** structure name must be the prefix appended with ‘info’. Also, the driver flag must be externally accessible. The flag name must be the prefix appended with ‘devflag’.

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

```

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

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

```

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

The `module_info` contains identification and limit values. All queues associated with a certain driver/module share the same `module_info` structures. The `module_info` structures define the characteristics of that driver/module's queues. As with the `qinit`, this structure is intended to be software read-only.

However, the four limit values (`q-minpsz`, `q-maxpsz`, `q-hiwat`, `q-lowat`) are copied to a queue structure where they are modifiable. In the example, the flow control high and low water marks are zero since there is no service procedure and messages are not queued in the module.

Three names are associated with a module: the character string in `fmodsw`, obtained from the name of the '`master.d`' file used to configure the module; the prefix for `streamtab`, used in configuring the module; and the module name field in the `module_info` structure. The module name must be the same as that of '`master.d`' for autoconfiguration. Each module ID and module name should be unique in the system. The module ID is currently used only in logging and tracing. It is '`0x08`' in the example.

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

The `module_stat` is optional. Currently, there is no *STREAMS* support for statistical information gathering.

7.1.1.1 Null Module Example

The null module procedures are as follows:

```
static int
modopen(q, devp, flag, sflag, credp)
    queue_t *q;      /* pointer to the read queue */
    dev_t *devp;     /* pointer to major/minor device number */
    int flag;        /* file flags */
    int sflag;       /* stream open flags */
    cred_t *credp;   /* pointer to a credentials structure */
{
    /* return success */
    return 0;
}

static int
modput(q, mp)        /* put procedure */
    queue_t *q;      /* pointer to the queue */
```

```

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

/* NOTE: we only need one put procedure that can be used for both
 * read-side and write-side.
 */
static int
modclose(q, flag, credp)
    queue_t *q;      /* pointer to the read queue */
    int flag;      /* file flags */
    cred_t *credp; /* pointer to a credentials structure */
{
    return 0;
}

```

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

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

MODOPEN	normal module open
0	normal driver open
CLONEOPEN	clone driver open

The return value from open is ‘0’ for success and an error number for failure. If a driver is called with the CLONEOPEN flag, the device number pointed to by the *devp* should be set by the driver to an unused device number accessible to that driver. This should be an entire device number (major and minor device number). The open procedure for a module is called on the first I_PUSH and on all subsequent open calls to the same *Stream*. During a push, a nonzero return value causes the I_PUSH to fail and the module to be removed from the *Stream*. If an error is returned by a module during an open call, the open fails, but the *Stream* remains intact.

The module open fails if not opened by the super-user (also referred to as a privileged user) that in future releases will be a user with ‘*driver/special*’ permissions. Permission checks in module and driver open routines should be done with the **drv_priv(9)** routine. For *UNIX System V Release 4.0*, there is no need to check if ‘*u.u_uid == 0*’. This and the **suser(9)** routine are replaced with:

```

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

```

In the null module example, **modopen** simply returns successfully. **modput** illustrates the common interface to put procedures. The arguments are the read or write queue pointer, as appropriate, and the message pointer. The put procedure in the appropriate side of the

queue is called when a message is passed from upstream or downstream. The put procedure has no return value. In the example, no message processing is performed. All messages are forwarded using the `putnext(9)` macro (see [Appendix C \[STREAMS Utilities\]](#), page 295). `putnext` calls the put procedure of the next queue in the proper direction.

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

7.2 STREAMS Input and Output Controls

STREAMS is an addition to the *UNIX* system traditional character input/output (I/O) mechanism. In this section, the phrases "*character I/O mechanism*" and "*I/O mechanism*" refer only to that part of the mechanism that pre-existed *STREAMS*.

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

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

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

The second difference is the absence of user context in a module and driver when the information associated with the `ioctl` is received. This prevents use of `copyin(9)` or `copyout(9)` by the module. This also prevents the module and driver from associating any kernel data with the currently running process. (It is likely that by the time the module or driver receives the `ioctl`, the process generating it may no longer be running.)

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

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

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

7.2.1 General `ioctl` Processing

STREAMS blocks a user process which issues an `ioctl` and causes the Stream head to generate an `M_IOCTL` message. The process remains blocked until either:

- a module or a driver responds with an `M_IOCACK` (ack, positive acknowledgement) message or an `M_IOCNAK` (nak, negative acknowledgement) message, or
- no message is received and the request "times out," or
- the `ioctl` is interrupted by the user process, or
- an error condition occurs.

For the `ioctl` `I_STR` the timeout period can be a user specified interval or a default. For the other `M_IOCTL` `ioctls`, the default value (infinite) is used.

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

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

The form of an `M_IOCTL` message is a single `M_IOCTL` message block followed by (see `M_PROTO` and `M_PCPROTO` Message Structure, see [Section B.1 \[Message Types\]](#), page 281) zero or more `M_DATA` blocks. The `M_IOCTL` message block contains an `iocblk` structure, defined in `'sys/stream.h'`:

```
struct iocblk {
    int ioc_cmd;           /* ioctl command type */
    cred_t *ioc_cr;        /* full credentials */
    uint ioc_id;           /* ioctl id */
    uint ioc_count;        /* count of bytes in data field */
    int ioc_error;         /* error code */
    int ioc_rval;          /* return value */
    long ioc_filler[4];    /* reserved for future use */
};

#define ioc_uid ioc_cr->cr_uid
#define ioc_gid ioc_cr->cr_gid
```

For an `I_STR` `ioctl`, `ioc_cmd` contains the command supplied by the user in the `striocctl` structure defined in `streamio(7)`.

If a module or driver determines an `M_IOCTL` message is in error for any reason, it must produce the negative acknowledgement message. This is typically done by setting the message type to `M_IOCNAK` and sending the message upstream. No data or a return value can be sent to a user in this case. If `ioc_error` is set to 0, the *Stream* head will cause the `ioctl` call to fail with `[EINVAL]`. The driver has the option of setting `ioc_error` to an alternate error number if desired.

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

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

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

7.2.2 I_STR ioctl Processing

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

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

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

The following example illustrates processing associated with an `I_STR` `ioctl`. `lpdoioctl` is called to process trapped `M_IOCTL` messages:

```
lpdoioctl(lp, mp)
struct lp *lp;
mblk_t *mp;

{
    struct iocblk *iocp;
    queue_t *q;

    q = 1 p->qptr;
```

```

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

switch (iocp->ioc_cmd) {
case SET_OPTIONS:
/* Count should be exactly one short's worth (for this example)
*/
if (iocp->ioc_count != sizeof(short))
    goto iocnak;
if (mp->b_cont == NULL)
    goto lognak; /* not shown in this example */
/* Actual data is in 2nd message block */
lpsetopt(lp, *(short *) mp->b_cont->b_rptr);
/* ACK the ioctl */
mp->b_datap->db_type = M_IOCACK;
iocp->ioc_count = 0;
greply(q, mp);
break;
default:
    iocnak:
/* NAK the ioctl */
mp->b_datap->db_type = M_IOCNAK;
greply(q, mp);
}
}

```

`lpdoioctl` illustrates driver `M_IOCTL` processing which also applies to modules. However, at case default, a module would not "nak" an unrecognized command, but would pass the message on. In this example, only one command is recognized, `SET_OPTIONS`. `ioc_count` contains the number of user supplied data bytes. For this example, it must equal the size of a short. The user data are sent directly to the printer interface using `lpsetopt`. Next, the `M_IOCTL` message is changed to type `M_IOCACK` and the `ioc_count` field is set to zero to indicate that no data are to be returned to the user. Finally, the message is sent upstream using `greply(9)`. If `ioc_count` was left nonzero, the Stream head would copy that many bytes from the 2nd Nth message blocks into the user buffer.

7.2.3 Transparent ioctl Processing

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

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

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

This mechanism is needed because user context does not exist in modules and drivers when `ioctl` processing occurs. This prevents them from using the kernel `copyin(9)/copyout(9)` functions. For example, consider the following `ioctl` call:

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

where `ioctl_struct` is a structure containing the members:

```
int stringlen;           /* string length */
char *string;
struct other_struct *other1;
```

To read (or write) the elements of `ioctl_struct`, a module would have to perform a series of `copyin(9)/copyout(9)` calls using pointer information from a prior `copyin(9)` to transfer additional data. A non *STREAMS* character driver could directly execute these copy functions because user context exists during all *UNIX* system calls to the driver. However, in *STREAMS*, user context is only available to modules and drivers in their `open` and `close` routines.

The transparent mechanism enables modules and drivers to request that the *Stream* head perform a `copyin(9)` or `copyout(9)` on their behalf to transfer `ioctl` data between their kernel space and various user space locations. The related data are sent in message pairs exchanged between the *Stream* head and the module. A pair of messages is required so that each transfer can be acknowledged. In addition to `M_IOCTL`, `M_IOCACK`, and `M_IOCNAK` messages, the transparent mechanism also uses `M_COPYIN`, `M_COPYOUT`, and `M_IOCDATA` messages.

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

1. The *Stream* head does not recognize the command argument of an `ioctl` call and creates a transparent `M_IOCTL` message (the `iocblk` structure has a `TRANSPARENT` indicator, see [Section 7.2.4 \[Transparent ioctl Messages\]](#), page 114) containing the value of the `arg` argument in the call. It sends the `M_IOCTL` message downstream.
2. A module receives the `M_IOCTL` message, recognizes the `ioc_cmd`, and determines that it is `TRANSPARENT`.
3. If the module requires user data, it creates an `M_COPYIN` message to request a `copyin(9)` of user data. The message will contain the address of user data to copy in and how much data to transfer. It sends the message upstream.
4. The *Stream* head receives the `M_COPYIN` message and uses the contents to `copyin(9)` the data from user space into an `M_IOCDATA` response message which it sends downstream. The message also contains an indicator of whether the data transfer succeeded (the `copyin(9)` might fail, for instance, because of an `[EFAULT]` [see `errno(3)`] condition).

5. The module receives the `M_IOCADATA` message and processes its contents. The module may use the message contents to generate another `M_COPYIN`. Steps 3 through 5 may be repeated until the module has requested and received all the user data to be transferred.
6. When the module completes its data transfer, it performs the `ioctl` processing and sends an `M_IOCACK` message upstream to notify the *Stream* head that `ioctl` processing has successfully completed.

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

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

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

7.2.4 Transparent `ioctl` Messages

The form of the `M_IOCTL` message generated by the *Stream* head for a transparent `ioctl` is a single `M_IOCTL` message block followed by one `M_DATA` block. The form of the `iocblk` structure in the `M_IOCTL` block is the same as described above see [Section 7.2.1 \[General `ioctl` Processing\]](#), page 110). However, `ioc_cmd` is set to the value of the command argument in the `ioctl` system call and `ioc_count` is set to `TRANSPARENT`, defined in '`sys/stream.h`'. `TRANSPARENT` distinguishes the case where an `I_STR` `ioctl` may specify a value of `ioc_cmd` equivalent to the command argument of a transparent `ioctl`. The `M_DATA` block of the message contains the value of the `arg` parameter in the call.

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

`M_COPYIN`, `M_COPYOUT`, and `M_IOCADATA` messages and their use are described in more detail (see [Section B.1 \[Message Types\]](#), page 281).

7.2.5 Transparent `ioctl` Examples

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

7.2.5.1 `M_COPYIN` Example

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

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

where `bufadd` is a structure declared as

```

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

```

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

```

struct address {          /* same members as in user space */
    int ad_len;           /* length in bytes */
    caddr_t ad_addr;      /* buffer address */
};

/* state values (overloaded in private field) */
#define GETSTRUCT 0       /* address structure */
#define GETADDR 1        /* byte string from ad_addr */

xxxwput(q, mp)
    queue_t *q;          /* write queue */
    mblk_t *mp;
{
    struct iocblk *iocbp;
    struct copyreq *cqp;

    switch (mp->b_datap->db_type) {
        .
        .
        .
        case M_IOCTL:
            iocbp = (struct iocblk *) mp->b_rptr;
            switch (iocbp->ioc_cmd) {
                case SET_ADDR:
                    if (iocbp->ioc_count != TRANSPARENT) {
                        /* fail if I_STR */
                        if (mp->b_cont) {
                            /* return buffer to pool ASAP */
                            freemsg(mp->b_cont);
                            mp->b_cont = NULL;
                        }
                        mp->b_datap->db_type = M_IOCNAK;          /* EINVAL */
                        greply(q, mp);
                        break;
                    }
                    /* Reuse M_IOCTL block for M_COPYIN request */
                    cqp = (struct copyreq *) mp->b_rptr;
                    /* Get user space structure address from linked M_DATA
                     block */
                    cqp->cq_addr = (caddr_t) *(long *) mp->b_cont->b_rptr;
                    freemsg(mp->b_cont);          /* MUST free linked blocks */
                    mp->b_cont = NULL;
                    cqp->cq_private = (mblk_t *) GETSTRUCT;
                    /* to identify response */
                    /* Finish describing M_COPYIN message */
                    cqp->cq_size = sizeof(struct address);

```

```

        cqp->cq_flag = 0;
        mp->b_datap->db_type = M_COPYIN;
        mp->b_wptr = mp->b_rptr + sizeof(struct copyreq);
        greply(q, mp);
        break;
    default:
        /* M_IOCTL not for us */
        /* if module, pass on */
        /* if driver, nak ioctl */
        break;
    }
    /* switch (iocbp->ioc_cmd) */
    break;
case M_IOCTLDATA:
    xxxioc(q, mp);
    break;
.
.
.
}
/* switch (mp->b_datap->db_type) */
}

```

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

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

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

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

```

xxxioc(q, mp)
    queue_t *q;
    mblk_t *mp;
{
    struct iocblk *iocbp;
    struct copyreq *cqp;
    struct copyresp *csp;
    struct address *ap;

    csp = (struct copyresp *) mp->b_rptr;
    iocbp = (struct iocblk *) mp->b_rptr;
    switch (csp->cp_cmd) {
        /* validate this M_IOCTLDATA is for this module */

    case SET_ADDR:
        if (csp->cp_rval) {
            /* GETSTRUCT or GETADDR failed */
            freemsg(mp);
            return;
        }
        switch ((int) csp->cp_private) {
            /* determine state */

```

```

case GETSTRUCT: /* user structure has arrived */
    mp->b_datap->db_type = M_COPYIN;    /* reuse M_IOCTLDATA
                                         block */

    cqp = (struct copyreq *) mp->b_rptr;
    /* user structure */
    ap = (struct address *) mp->b_cont->b_rptr;
    cqp->cq_size = ap->ad_len; /* buffer length */
    cqp->cq_addr = ap->ad_addr; /* user space buffer address */
    freemsg(mp->b_cont);
    mp->b_cont = NULL;
    cqp->cq_flag = 0;
    csp->cp_private = (mblk_t *) GETADDR;    /* next state */
    qreply(q, mp);
    break;

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

default:    /* invalid state: can't happen */
    freemsg(mp->b_cont);
    mp->b_cont = NULL;
    mp->b_datap->db_type = M_IOCNAK;
    mp->b_wptr = mp->b_rptr + sizeof(struct iocblk);
    iocbp->ioc_error = EINVAL; /* may have been overwritten */
    qreply(q, mp);
    ASSERT(0);    /* panic if debugging mode */
    break;
}
break;    /* switch (cp_private) */
default:    /* M_IOCTLDATA not for us */
    /* if module, pass message on */
    /* if driver, free message */
    break;
}    /* switch (cp_cmd) */
}

```

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

7.2.5.2 M_COPYOUT Example

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

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

or, alternately, by use of a `streamio(7)` call

```
ioctl(fd, I_, &opts_striocntl)
```

In the first case, `optadd` is declared `struct options`. In the `I_STR` case, `opts_striocntl` is declared `struct striocntl` where `opts_striocntl.ic_dp` points to the user `options` structure.

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

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

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

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

            switch (iocbp->ioc_cmd) {
                case GET_OPTIONS:
                    if (iocbp->ioc_count == TRANSPARENT) {
                        transparent = 1;
                        cqp = (struct copyreq *) mp->b_rptr;
                        cqp->cq_size = sizeof(struct options);
                        /* Get structure address from linked M_DATA block */
                        cqp->cq_addr = (caddr_t) *(long *) mp->b_cont->b_rptr;
                        cqp->cq_flag = 0;
                        /* No state necessary - we will only ever get one
                           M_IOCTLDATA from the Stream head indicating success or
                           failure for the copyout */
                    }
                    if (mp->b_cont)
```



```

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

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

case M_IOCDATA:
    csp = (struct copyresp *) mp->b_rptr;
    if (csp->cmd != GET_OPTIONS) { /* M_IOCDATA not for us */
        /* if module, pass on; if driver, free message */
        break;
    }
    if (csp->cp_rval) {
        freemsg(mp);    /* failure */
        return;
    }
    /* Data successfully copied out, ack */
    /* reuse M_IOCDATA for ack */
    mp->b_datap->db_type = M_IOCACK;
    mp->b_wptr = mp->b_rptr + sizeof(struct iocblk);
    iocbp->ioc_error = 0;    /* may have been overwritten */
    iocbp->ioc_count = 0;    /* may have been overwritten */
    iocbp->ioc_rval = 0;    /* may have been overwritten */
    qreply(q, mp);
    break;
}
/* switch (mp->b_datap->db_type) */
}

```

7.2.5.3 Bidirectional Transfer Example

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

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

```
ioctl(fd, XXX_IOCTL, &addr_xxxdata)
```

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

```
struct xxxdata {          /* same members in user space */
    int x_inlen;           /* number of bytes copied in */
    caddr_t x_inaddr;      /* buffer address of data copied in */
    int x_outlen;          /* number of bytes copied out */
    caddr_t x_outaddr;     /* buffer address of data copied out */
};

/* State information for ioctl processing */
struct state {
    int st_state;          /* see below */
    struct xxxdata st_data; /* see above */
};

/* state values */
#define GETSTRUCT          0      /* get xxxdata structure */
#define GETINDATA          1      /* get data from x_inaddr */
#define PUTOUTDATA         1      /* get response from M_COPYOUT */

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

    switch (mp->b_datap->db_type) {
        .
        .
        .
    case M_IOCTL:
        iocbp = (struct iocblk *) mp->b_rptr;
        switch (iocbp->ioc_cmd) {
            case XXX_IOCTL:
                if (iocbp->ioc_cont != TRANSPARENT) {
                    /* fail if I_STR */
                    if (mp->b_cont) {
                        /* return buffer to pool ASAP */
                        freemsg(mp->b_cont);
                        mp->b_cont = NULL;
                    }
                    mp->b_datap->db_type = M_IOCNAK;          /* EINVAL */
                    qreply(q, mp);
                    break;
                }
                /* Reuse M_IOCTL block for M_COPYIN request */
            }
        }
    }
}
```

```

    cqp = (struct copyreq *) mp->b_rptr;
    /* Get structure's user address from linked M_DATA block */
    cqp->cq_addr = (caddr_t) *(long *) mp->b_cont->b_rptr;
    freemsg(mp->b_cont);
    mp->b_cont = NULL;
    /* Allocate state buffer */
    if ((tmp =
        allocb(sizeof(struct state), BPRI_MED)) == NULL) {
        mp->b_datap->db_type = M_IOCCK;
        iocbp->ioc_error = EAGAIN;
        qreply(q, mp);
        break;
    }
    tmp->b_wptr += sizeof(struct state);
    stp = (struct state *) tmp->b_rptr;
    stp->st_state = GETSTRUCT;
    cqp->cq_private = tmp;
    /* Finish describing M_COPYIN message */
    cqp->cq_size = sizeof(struct xxxdata);
    cqp->cq_flag = 0;
    mp->b_datap->db_type = M_COPYIN;
    mp->b_wptr = mp->b_rptr + sizeof(struct copyreq);
    qreply(q, mp);
    break;

default:
    /* M_ioctl not for us */
    /* if module, pass on */
    /* if driver, nak ioctl */
    break;
}
break;
case M_IOCCK:
    xxxioc(q, mp);
    /* all M_IOCCK processing done here */
    break;
.
.
.
}
/* switch (mp->b_datap->db_type) */
}

```

xxxwput allocates a message block to contain the state structure and reuses the M_IOCTL to create an M_COPYIN message to read in the xxxdata structure.

M_IOCCK processing is done in xxxioc:

```

xxxioc(q, mp)
    queue_t *q;
    mblk_t *mp;
{
    struct iocblk *iocbp;
    struct copyreq *cqp;
    struct copyresp *csp;
    struct state *stp;
    mblk_t *xxx_indata();

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

```

```

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

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

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

case GETINDATA: /* data successfully copied in */
    /* Process input, return output */
    if ((mp->b_cont = xxx_indata(mp->b_cont)) == NULL) {
        /* hypothetical */
        mp->b_datap->db_type = M_IOCNAK;
        /* fall xxx_indata */
        mp->b_wptr = mp->b_rptr + sizeof(struct iocblk);
        iocbp->ioc_error = EIO;
        greply(q, mp);
        break;
    }
    mp->b_datap->db_type = M_COPYOUT;
    cqp = (struct copyreq *) mp->b_rptr;
    cqp->cq_size =
        min(msgdsize(mp->b_cont), stp->st_data.x_outlen);
    cqp->cq_addr = stp->st_data.x_outaddr;
    cqp->cq_flag = 0;
    stp->st_state = PUTOUTDATA; /* next state */
    greply(q, mp);
    break;

case PUTOUTDATA: /* data successfully copied out, ack
                    ioctl */
    freemsg(csp->cp_private); /* state structure */
    mp->b_datap->db_type = M_IOCACK;
    mp->b_wptr = mp->b_rptr + sizeof(struct iocblk);
    iocbp->ioc_error = 0; /* may have been overwritten */
    iocbp->ioc_count = 0; /* may have been overwritten */

```

```

        iocbp->ioc_rval = 0;          /* may have been overwritten */
        greply(q, mp);
        break;

    default:                          /* invalid state: can't happen */
        freemsg(mp->b_cont);
        mp->b_cont = NULL;
        mp->b_datap->db_type = M_IOCNAK;
        mp->b_wptr = mp->b_rptr + sizeof(struct iocblk);
        iocbp->ioc_error = EINVAL;
        greply(q, mp);
        ASSERT(0);                  /* panic if debugging mode */
        break;
    }                                /* switch (stp->st_state) */
    break;
default:                            /* M_IOCDATA not for us */
    /* if module, pass message on */
    /* if driver, free message */
    break;
}                                    /* switch (csp->cp_cmd) */
}

```

At case `GETSTRUCT`, the `xxxdata` structure is copied into the module's state structure (pointed at by `cp_private` in the message) and the `M_IOCDATA` message is reused to create a second `M_COPYIN` message to read in the user data. At case `GETINDATA`, the input user data are processed by the `xxx_indata` routine (not supplied in the example) which frees the linked `M_DATA` block and returns the output data message block. The `M_IOCDATA` message is reused to create an `M_COPYOUT` message to write the user data. At case `PUTOUTDATA`, the message block containing the state structure is freed and an acknowledgement is sent upstream.

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

7.2.6 I_LIST ioctl

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

The `strchg(1)` command does the following:

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

The `strconf(1)` command does the following:

- Indicate if the specified module is present on the *Stream*.
- Print the topmost module of the *Stream*.

- Print a list of all modules and topmost driver on the *Stream*.

If the *Stream* contains a multiplexing driver, the `strchg(1)` and `strconf(1)` commands will not recognize any modules below that driver.

The `ioctl I_LIST` performs two functions. When the third argument of the `ioctl` call is set to 'NULL', the return value of the call indicates the number of modules, including the driver, present on the *Stream*. For example, if there are two modules above the driver, 3 is returned. On failure, `errno` may be set to a value specified in `streamio(7)`. The second function of the `I_LIST` `ioctl` is to copy the module names found on the *Stream* to the user supplied buffer. The address of the buffer in user space and the size of the buffer are passed to the `ioctl` through a structure `str_list` that is defined as:

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

where `sl_nmods` is the number of modules in the `sl_modlist` array that the user has allocated. Each element in the array must be at least 'FMNAMESZ+1' bytes long. FMNAMESZ is defined by 'sys/conf.h'.

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

It is possible, but unlikely, that another module was pushed on the *Stream* after the user invoked the `I_LIST` `ioctl` with the 'NULL' argument and before the `I_LIST` `ioctl` with the structure argument was invoked.

7.3 STREAMS Flush Handling

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

FLUSHR	Flush read queue.
FLUSHW	Flush write queue.
FLUSHRW	Flush both, read and write, queues.
FLUSHBAND	Flush a specified priority band only.

The following example shows line discipline module flush handling:

```

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

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

        if (*mp->b_rptr & FLUSHR)      /* flush read queue */
            flushq((q->q_flag & QREADR) ? q : RD(q), FLUSHDATA);
        putnext(q, mp);      /* pass it on */
        return;
    }
}

```

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

The next example shows the line discipline module flushing due to break:

```

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

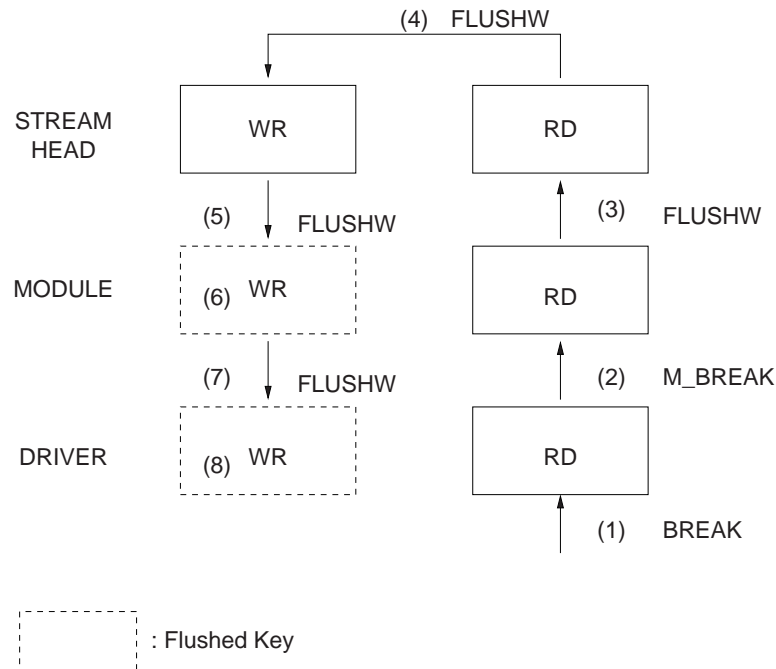
    case M_FLUSH:
        if (*mp->b_rptr & FLUSHW)      /* flush write queue */
            flushq((q->q_flag & QREADR) ? WR(q) : q, FLUSHDATA);

        if (*mp->b_rptr & FLUSHR)      /* flush read queue */
            flushq((q->q_flag & QREADR) ? q : RD(q), FLUSHDATA);
        putnext(q, mp);      /* pass it on */
        return;

    case M_BREAK:
        if (q->q_flag & QREADR) {      /* read side only */
            /* it doesn't make sense for write side */
            putctl1(q->q_next, M_PCSIG, SIGINT);
            putctl1(q->q_next, M_FLUSH, FLUSHW);
            putctl1(WR(q)->q_next, M_FLUSH, FLUSHR);
        }
        return;
    }
}

```

The next two figures further demonstrate flushing the entire *Stream* due to a line break. [Figure 7.1](#) shows the flushing of the write-side of a *Stream*, and [Figure 7.2](#) shows the flushing of the read-side of a *Stream*. In the figures dotted boxes indicate flushed queues.

Figure 7.1: *Flushing the Write-Side of a Stream*

The following takes place:

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

This figure shows flushing read-side of a *Stream*.

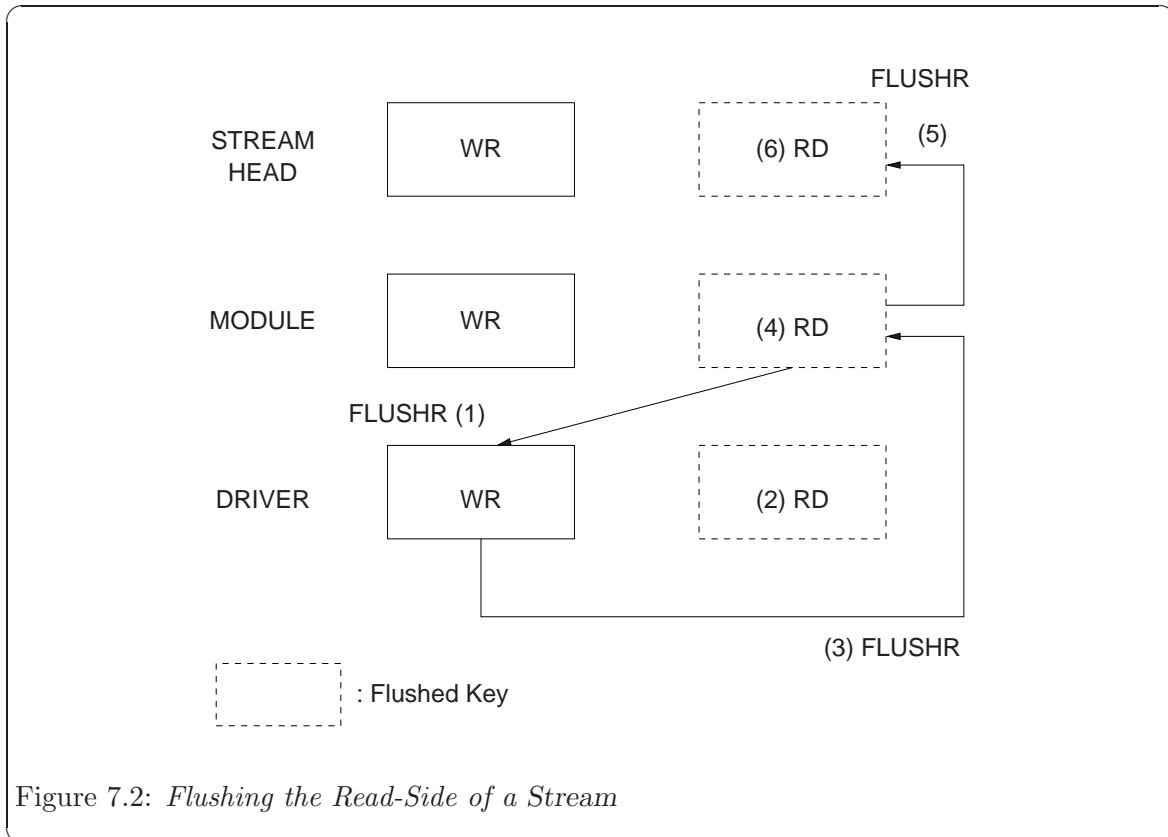


Figure 7.2: *Flushing the Read-Side of a Stream*

The events taking place are:

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

The `flushband(9)` routine (see [Appendix C \[STREAMS Utilities\]](#), page 295) provides the module and driver with the capability to flush messages associated with a given priority band. A user can flush a particular band of messages by issuing:

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

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

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

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

The following example shows flushing according to the priority band:

```

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

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

```

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

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

7.4 STREAMS Driver-Kernel Interface

The *Driver-Kernel Interface (DKI)* is an interface between the *UNIX* system kernel and drivers. These drivers are block interface drivers, character interface drivers, and drivers and modules supporting a *STREAMS* interface. Each driver type supports an interface from the kernel to the driver. This kernel to-driver interface consists of a set of driver-defined functions that are called by the kernel. These functions are the entry points into the driver.

One benefit of defining the *DKI* is increased portability of driver source code between various *UNIX System V* implementations. Another benefit is a gain in modularity that results in extending the potential for changes in the kernel without breaking driver code.

The interaction between a driver and the kernel can be described as occurring along two paths. (See [Figure 7.3](#)).

One path includes those functions in the driver that are called by the kernel. These are entry points into the driver. The other path consists of the functions in the kernel that are called by the driver. These are kernel utility functions used by the driver. Along both paths, information is exchanged between the kernel and drivers in the form of data structures. The *DKI* identifies these structures and specifies a set of contents for each. The *DKI* also defines the common set of entry points expected to be supported in each driver type and their calling and return syntaxes. For each driver type, the *DKI* lists a set of kernel utility functions that can be called by that driver and also specifies their calling and return syntaxes.

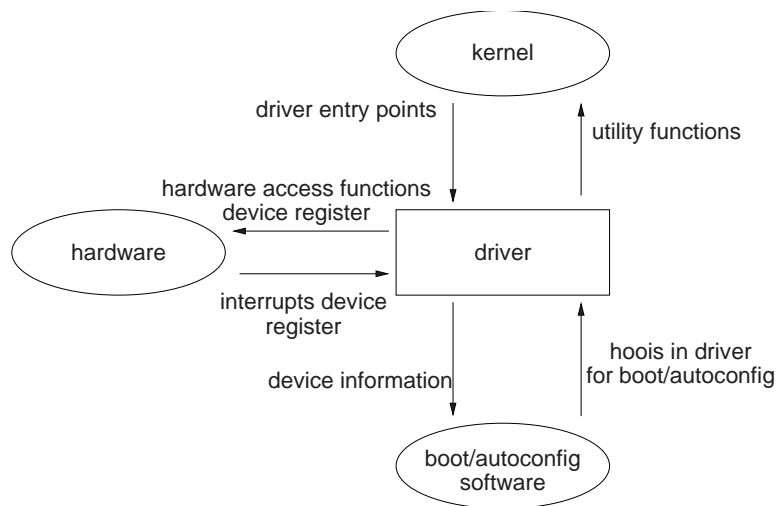


Figure 7.3: *Interfaces Affecting Drivers*

The set of *STREAMS* utilities available to drivers are listed (see [Appendix C \[STREAMS Utilities\]](#), page 295). No system-defined macros that manipulate global kernel data or introduce structure size dependencies are permitted in these utilities. Therefore, some utilities that have been implemented as macros in the prior *UNIX* system releases are implemented as functions in *UNIX System V Release 4.0*. This does not preclude the existence of both macro and function versions of these utilities. It is envisioned that driver source code will include a header file (see [Section 7.5.3 \[Header Files\]](#), page 135, later in this chapter) that picks up function declarations while the core operating system source includes a header file that defines the macros. With the *DKI* interface the following *STREAMS* utilities are implemented as ‘C’ programming language functions: `datamsg(9)`, `OTHERQ(9)`, `putnext(9)`, `RD(9)`, `splstr(9)`, and `WR(9)`.

Replacing macros such as `RD(9)` with function equivalents in the driver source code allows driver objects to be insulated from changes in the data structures and their size, further increasing the useful lifetime of driver source code and objects.

The *DKI* interface defines an interface suitable for drivers and there is no need for drivers to access global kernel data structures directly. The kernel functions `drv_getparm(9)` and `drv_setparm(9)` are provided for reading and writing information in these structures. This restriction has an important consequence. Since drivers are not permitted to access global kernel data structures directly, changes in the contents/offsets of information within these structures will not break objects.

7.4.1 Device Driver Interface and Driver-Kernel Interface

The *Device Driver Interface (DDI)* is an *AT&T* interface that facilitates driver portability across different *UNIX* system versions on the *AT&T 3B2* hardware. The *Driver-Kernel Interface (DKI)* is an interface that also facilitates driver source code portability across implementations of *UNIX System V Release 4.0* on all machines. *DKI* driver code, however, will have to be recompiled on the machine on which it is to run.

The most important distinction between the *DDI* and the *DKI* lies in scope. The *DDI* addresses complete interfaces (see note below) for block, character, and *STREAMS* interface drivers and modules. The *DKI* defines only driver interfaces with the kernel with the addition of the kernel interface for file system type (*FST*) modules. The *DKI* interface does not specify the system initialization driver interface [i.e., `init()` and `start()` driver routines] nor hardware related interfaces such as `getvec` for the *AT&T 3B2*.

The "complete interface" refers to hardware- and boot/configuration-related driver interface in addition to the interface with the kernel.

7.4.2 STREAMS Interface

The entry points from the kernel into *STREAMS* drivers and modules are through the `qinit` structures (see [Appendix A \[STREAMS Data Structures\], page 275](#)) pointed to by the `streamtab` structure, `prefixinfo`. *STREAMS* drivers may need to define additional entry points to support the interface with boot/autoconfiguration software and the hardware (for example, an interrupt handler).

If the *STREAMS* module has prefix *mod* then the declaration is of the form:

```
static int modwput(), modwput(), modwsrv(), modclose();

static int modwput(), modwput(), modwsrv();

static struct qinit rdinit =
    { modrput, modrsrv, modopen, modclose, NULL, struct module_info,
      NULL };

static struct qinit wdinit =
    { modwput, modwsrv, NULL, NULL, NULL, struct module_info, NULL };

struct streamtab modinfo = { &rdinit, &wrinit, NULL, NULL };

extern int moddevflag = 0;
```

where `modrput` is the module's read queue put procedure, `modrsrv` is the module's read queue service procedure, `modopen` is the open routine for the module, `modclose` is the close routine for the module, `modwput` is the put procedure for the module's write queue, and `modwsrv` is the service procedure for the module's write queue.

Each `qinit` structure can point to four entry points. (An additional function pointer has been reserved for future use and must not be used by drivers or modules.) These four function pointer fields in the `qinit` structure are: `qi_putp`, `qi_srvp`, `qi_qopen`, and `qi_close`. The utility functions that can be called by *STREAMS* drivers and modules are listed (see [Appendix C \[STREAMS Utilities\]](#), page 295). They must follow the call and return syntaxes specified in the appendix.

7.5 STREAMS Design Guidelines

This section summarizes guidelines common to the design of *STREAMS* modules and drivers. Additional rules pertaining to modules and drivers can be found in [Section 13.2 \[Modules\]](#), page 225, for modules and [Section 13.3 \[Drivers\]](#), page 225, for drivers.

7.5.1 Module and Driver Rules

1. Modules and drivers cannot access information in the `current task_struct` structure of a process. Modules and drivers are not associated with any process, and therefore have no concept of process or user context, except during `open` and `close` routines (see [Section 7.5.1.1 \[Rules for Open and Close Routines\]](#), page 131).
2. Every module and driver must process an `M_FLUSH` message according to the value of the argument passed in the message.
3. A module or a driver should not change the contents of a data block whose reference count is greater than 1 [see `dupmsg(9)`] because other modules/drivers that have references to the block may not want the data changed. To avoid problems, data should be copied to a new block and then changed in the new one.
4. Modules and drivers should manipulate queues and manage buffers only with the routines provided for that purpose, (see [Appendix C \[STREAMS Utilities\]](#), page 295).
5. Modules and drivers should not require the data in an `M_DATA` message to follow a particular format, such as a specific alignment.
6. Care must be taken when modules are mixed and matched, because one module may place different semantics on the priority bands than another module. The specific use of each band by a module should be included in the service interface specification. When designing modules and drivers that make use of priority bands one should keep in mind that priority bands merely provide a way to impose an ordering of messages on a queue. The priority band is not used to determine the service primitive. Instead, the service interface should rely on the data contained in the message to determine the service primitive.

7.5.1.1 Rules for Open and Close Routines

1. `open` and `close` routines may sleep, but the sleep must return to the routine in the event of a signal. That is, if they sleep, they must be at '`priority <= PZERO`', or with '`PCATCH`' set in the sleep priority.
2. The `open` routine should return zero on success or an error number on failure. If the `open` routine is called with the `CLONEOPEN` flag, the device number should be set by the

driver to an unused device number accessible to that driver. This should be an entire device number (major/minor).

3. `open` and `close` routines have user context and can access the `current task_struct` structure. Only the following fields can be accessed in the `u_area` (`'user.h'`): `u_procp`, `u_ttyp`, `u_uid`, `u_gid`, `u_ruid`, and `u_rgid`. The fields `u_uid`, `u_gid`, `u_ruid`, and `u_rgid` are for backward compatibility with previously designed device drivers. The actual user credentials are passed directly to the driver and need not be accessed in the `current task_struct` structure. These fields may not support valid `uids` or `gids` when the system is configured with large user ids. See note.
4. Only the following fields can be accessed in the process table (`'proc.h'`): `p_pid`, `p_pgrp`. See note.
5. If a module or a driver wants to allocate a controlling terminal, it should send an `M_SETOPTS` message to the *Stream* head with the `SO_ISTTY` flag set. Otherwise signaling will not work on the *Stream*.

The *DKI* interface provides the `drv_getparm(9)` and `drv_setparm(9)` functions to read/write these data and the driver/module need not access them directly.

7.5.1.2 Rules for Input Output Controls

- Do not change the `ioc_id`, `ioc_uid`, `ioc_gid`, or `ioc_cmd` fields in an `M_IOCTL` message.
- The above rule also applies to fields in an `M_IOCTLDATA`, `M_COPYIN`, and `M_COPYOUT` message. (Field names are different; see [Appendix A \[STREAMS Data Structures\]](#), [page 275](#))
- Always validate `ioc_count` to see whether the `ioctl` is the *transparent* or `I_STR` form.

7.5.1.3 Rules for Put and Service Procedures

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

- `Put` and `service` procedure must not sleep.
- Return codes can be sent with *STREAMS* messages `M_IOCACK`, `M_IOCNAK`, and `M_ERROR`.
- Protect data structures common to `put` and `service` procedures by using `splstr(9)`.
- `Put` and `service` procedures cannot access the information in the `current task_struct` structure of a process.
- Processing `M_DATA` messages by both `put` and `service` procedures could lead to messages going out of sequence. The `put` procedure should check if any messages were queued before processing the current message.

On the read-side, it is usually a good idea to have the `put` procedure check if the `service` procedure is running because of the possibility of a race condition. That is, if there are unprotected sections in the `service` procedure, the `put` procedure can be called and run to completion while the `service` procedure is running (the `put` procedure can interrupt the `service` procedure on the read-side). For example, the `service` procedure is running and it removes the last message from the queue, but before it puts the message upstream

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

Put Procedures

1. Each queue must define a **put** procedure in its **qinit** structure for passing messages between modules.
2. A **put** procedure must use the **putq(9)** (see [Appendix C \[STREAMS Utilities\]](#), [page 295](#)) utility to enqueue a message on its own queue. This is necessary to ensure that the various fields of the queue structure are maintained consistently.
3. When passing messages to a neighboring module, a module may not call **putq(9)** directly, but must call its neighbor module's **put** procedure [see **putnext(9)**]. However, the *q_qinfo* structure that points to a module's **put** procedure may point to **putq(9)** [i.e., **putq(9)** is used as the **put** procedure for that module]. When a module calls a neighbor module's **put** procedure that is defined in this manner, it will be calling **putq(9)** indirectly. If any module uses **putq(9)** as its **put** procedure in this manner, the module must define a **service** procedure. Otherwise, no messages will ever be processed by the next module. Also, because **putq(9)** does not process **M_FLUSH** messages, any module that uses **putq(9)** as its **put** procedure must define a **service** procedure to process **M_FLUSH** messages.
4. The **put** procedure of a queue with no **service** procedure must call the **put** procedure of the next queue using **putnext(9)**, if a message is to be passed to that queue.
5. Processing many function calls with the **put** procedure could lead to interrupt stack overflow. In that case, switch to **service** procedure processing whenever appropriate to switch to a different stack.

Service Procedures

1. If flow control is desired, a **service** procedure is required. The **canput(9)** or **bcanput(9)** routines should be used by **service** procedures before doing **putnext(9)** to honor flow control.
2. The **service** procedure must use **getq(9)** to remove a message from its message queue, so that the flow control mechanism is maintained.
3. The **service** procedure should process all messages on its queue. The only exception is if the *Stream head* is blocked [i.e., **canput(9)** fails] or some other failure like buffer allocation failure. Adherence to this rule is the only guarantee that *STREAMS* will enable (schedule for execution) the **service** procedure when necessary, and that the flow control mechanism will not fail. If a **service** procedure exits for other reasons, it must take explicit steps to assure it will be re-enabled.
4. The **service** procedure should not **put** a high priority message back on the queue, because of the possibility of getting into an infinite loop.

5. The **service** procedure must follow the steps below for each message that it processes. *STREAMS* flow control relies on strict adherence to these steps.

- Step 1: Remove the next message from the queue using **getq(9)**. It is possible that the **service** procedure could be called when no messages exist on the queue, so the **service** procedure should never assume that there is a message on its queue. If there is no message, return.
- Step 2: If all of the following conditions are met: **canput(9)** or **bcanput(9)** fails and the message type is not a high priority type and the message is to be **put** on the next queue, continue at Step 3. Otherwise, continue at Step 4.
- Step 3: The message must be replaced on the head of the queue from which it was removed using **putbq(9)** (see [Appendix C \[STREAMS Utilities\]](#), page 295). Following this, the **service** procedure is exited. The **service** procedure should not be re-enabled at this point. It will be automatically back-enabled by flow control.
- Step 4: If all of the conditions of Step 2 are not met, the message should not be returned to the queue. It should be processed as necessary. Then, return to Step 1.

7.5.2 STREAMS Data Structures

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

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

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

7.5.2.1 Dynamic Allocation of STREAMS Data Structures

Prior to *UNIX System V Release 4.0*, *STREAMS* data structures were statically configured to support a fixed number of *Streams*, read and write queues, message and data blocks, link block data structures, and *Stream* event cells. The only way to change this configuration was to reconfigure and reboot the system. Resources were also wasted because data structures were allocated but not necessarily needed.

With *Release 4.0* the *STREAMS* mechanism has been enhanced to dynamically allocate the following *STREAMS* data structures: **stdata**, **queue**, **linkblk**, **strevent**, **datab**, and **msgb**. *STREAMS* allocates memory to cover these structures as needed.

Dynamic data structure allocation has the advantage of the kernel being initially smaller than a system with static configuration. The performance of the system may also improve because of better memory utilization and added flexibility. However, **allocb(9)**, **bufcall(9)**, and **freeb(9)**, the routines that manage these data structures, may be slower at times because of extra overhead needed for dynamic allocation.

7.5.3 Header Files

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

- ‘types.h’ contains type definitions used in the *STREAMS* header files
- ‘stream.h’
contains required structure and constant definitions
- ‘stropts.h’
primarily for users, but contains definitions of the arguments to the `M_FLUSH` message type also required by modules
- ‘ddi.h’ contains definitions and declarations needed by drivers to use functions for the *UNIX System V Device Driver Interface* or *Driver-Kernel Interface*. This header file should be the last header file included in the driver source code (after all statements).

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

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

7.5.4 Accessible Symbols and Functions

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

- ‘user.h’ (from open/close procedures only)
 - `u_procp` process structure pointer
 - `u_ttyp` tty group ID pointer
- ‘proc.h’ (from open/close procedures only)
 - `p_pid` process ID
 - `p_pgrp` process group ID

'(none)'	functions accessible from open/close procedures only
	<code>sleep(chan, pri)</code> sleep until wakeup
	<code>delay(ticks)</code> delay for a specified time
'(none)'	universally accessible functions
	<code>bcopy(from, to, nbytes)</code> copy data quickly
	<code>bzero(buffer, nbytes)</code> zero data quickly
	<code>max(a, b)</code> return max of args
	<code>min(a, b)</code> return min of args
	<code>rmalloc(mp, size)</code> allocate memory space
	<code>rmfree(mp, size, i)</code> de-allocate memory space
	<code>rminit(mp, mapsize)</code> initialize map structure
	<code>vtop(vaddr, NULL)</code> translate from virtual to physical address
	<code>cmn_err(level, ...)</code> print message and optional panic
	<code>spln()</code> set priority level
	<code>splstr()</code> set processor level for Streams
	<code>timeout(func, arg, ticks)</code> schedule event
	<code>untimeout(id)</code> cancel event
	<code>wakeup(chan)</code> wake up sleeper
'sysmacros.h'	The first four functions are used to get the major/minor part of the expanded device number.
	<code>getemajor(x)</code> return external major part
	<code>getmajor(x)</code> return internal major part

<code>getemisor(x)</code>	return external minor part
<code>getminor(x)</code>	return internal minor part
<code>makedev(x, y)</code>	create a old device number
<code>makedevice(x, y)</code>	create a new device number
<code>cmpdev(x)</code>	convert to old device format
<code>expdev(x)</code>	convert to new device format
<code>'system.h'</code>	system file
<code>lbolt</code>	clock ticks since boot in HZ
<code>time</code>	seconds since epoch
<code>'param.h'</code>	parameter file
<code>PZERO</code>	zero sleep priority
<code>PCATCH</code>	catch signal sleep flag
<code>HZ</code>	clock ticks per second
<code>NULL</code>	0
<code>'types.h'</code>	Everything in <code>'types.h'</code> can be used.

8 STREAMS Modules

8.1 Modules

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

8.1.1 Module Routines

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

A module's `put` routine is called by the preceding module, driver, or *Stream head* and before the corresponding `service` routine. The `put` routine should do any processing that needs to be done immediately (for example, processing of high priority messages). Any processing that can be deferred should be left for the corresponding `service` routine.

The `service` routine is used to implement flow control, handle de-packetization of messages, perform deferred processing, and handle resource allocation. Once the `service` routine is enabled, it always runs before any user level code. The `put` and `service` routines must not call `sleep(9)` and cannot access the `current task_struct` structure, because they are executed asynchronously with respect to any process.

The following example shows a *STREAMS* module read-side `put` routine:

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

    modptr = (struct mod_prv *) q->q_ptr;      /* for state
                                                information */
    if (mp->b_datap->db_type >= QPCTL) {        /* process priority
                                                message */
        putnext(q, mp);      /* and pass it on */
        return;
    }
    switch (mp->b_datap->db_type) {
        case M_DATA;          /* may process message data */
            putq(q, mp);      /* queue message for service routine */
            return;
        case M_PROTO;         /* handle protocol control message */
        ...default:
            putnext(q, mp);
            return;
    }
}
```

The following briefly describes the code:

- A pointer to a queue defining an instance of the module and a pointer to a message are passed to the `put` routine.
- The `put` routine switches on the type of the message. For each message type, the `put` routine either enqueues the message for further processing by the module `service` routine, or passes the message to the next module in the *Stream*.
- High priority messages are processed immediately by the `put` routine and passed to the next module.
- Ordinary (or normal) messages are either enqueued or passed along the *Stream*.

This example shows a module write-side `put` routine:

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

    modptr = (struct mod_prv *) q->q_ptr;      /* for state
                                                information */
    if (mp->b_datap->db_type >= QPCTL) {        /* process priority
                                                message */
        putnext(q, mp);      /* and pass it on */
        return;
    }
    switch (mp->b_datap->db_type) {
    case M_DATA:      /* may process message data */
        putq(q, mp);  /* queue message for service routine */
        /* or pass message along */
        /* putnext(q, mp); */
        return;
    case M_PROTO:
        .
        .
        .
    case M_IOCTL:      /* if command in message is
                        recognized */
        /* process message and send back reply */
        /* else pass message downstream */
    default:
        putnext(q, mp);
        return;
    }
}
```

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

The following example shows a general scenario employed by the module's `service` routine:

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

```

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

```

The steps are:

- Retrieve the first message from the queue using `getq(9)`. If the message is high priority, process
- it immediately, and pass it along the *Stream*.
- Otherwise, the **service** routine should use the `canput(9)` utility to determine if the next module or driver that enqueues messages is within acceptable flow control limits. The `canput(9)` routine goes down (or up on the read-side) the *Stream* until it reaches a module with a **service** routine, a driver, or the *Stream head*. When it reaches one, it looks at the total message space currently allocated at that queue for enqueued messages. If the amount of space currently used at that queue exceeds the high water mark, the `canput(9)` routine sets the QWANTW flag¹ for the queue and returns ‘false’ (zero). If the next queue with a **service** routine is within acceptable flow control limits, `canput(9)` simply returns ‘true’ (nonzero).
- If `canput(9)` returns ‘false’, the **service** routine should return the message to its own queue using the `putbq(9)` routine. The **service** routine can do no further processing at this time, and it should return.
- If `canput(9)` returns ‘true’, the **service** routine should complete any processing of the message. This may involve retrieving more messages from the queue, (de)-allocating header and trailer information, and performing control function for the module.
- When the **service** routine is finished processing the message, it may call the `putnext(9)` routine to pass the resulting message to the next queue.
- Above steps are repeated until there are no messages left on the queue (that is, `getq(9)` returns ‘NULL’) or `canput(9)` returns ‘false’.

8.1.2 Filter Module Example

The module shown next, `crmod`, is an asymmetric filter. On the write-side, *newline* is converted to *carriage return* followed by *newline*. On the read-side, no conversion is done. The

¹ Setting the QWANTW flag on the queue is an indication to the `getq(9)` utility that the queue needs to be back-enabled when the count falls to the low water mark.

declarations of this module are essentially the same as those of the null module presented in the previous chapter:

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

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

static struct module_info minfo =
    { 0x09, "crmod", 0, INFPSZ, 512, 128 };

static int modopen(), modrput(), modwput(), modwsrv(), modclose();

static struct qinit rinit = {
    modrput, NULL, modopen, modclose, NULL, &minfo, NULL
};

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

struct streamtab crmdinfo = { &rinit, &winit, NULL, NULL };

extern int moddevflag = 0;
```

A ‘master.d’ file to configure crmod is shown in [Appendix E \[STREAMS Configuration\]](#), page 323. ‘sys/stropts.h’ includes definitions of flush message options common to user level, modules and drivers. modopen and modclose are unchanged from the null module example shown in [Chapter 7 \[Overview of STREAMS Modules and Drivers\]](#), page 105. modrput is like modput from the null module.

Note that, in contrast to the null module example, a single module_info structure is shared by the read-side and write-side. The module_info includes the flow control high and low water marks (512 and 128) for the write queue. (Though the same module_info is used on the read queue side, the read-side has no service procedure so flow control is not used.) The qinit contains the service procedure pointer.

The write-side put procedure, the beginning of the service procedure, and an example of flushing a queue are shown next:

```
static int
modwput(q, mp)
    queue_t *q;
    register mblk_t *mp;
{
    if (mp->b_datap->db_type >= QPCTL && mp->b_datap - &gt;
        db_type != M_FLUSH)
        putnext(q, mp);
    else
        putq(q, mp);          /* Put it on the queue */
}

static int
modwsrv(q)
```



```

        queue_t *q;
    {
        mblk_t *mp;

        while ((mp = getq(q)) != NULL) {
            switch (mp->b_datap->db_type) {
            default:
                if (canput(q->q_next)) {
                    putnext(q, mp);
                    break;
                } else {
                    putbq(q, mp);
                    return;
                }

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

```

modwput, the write **put** procedure, switches on the message type. High priority messages that are not type `M_FLUSH` are putnext to avoid scheduling. The others are queued for the **service** procedure. An `M_FLUSH` message is a request to remove messages on one or both queues. It can be processed in the **put** or **service** procedure.

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

For an `M_FLUSH` message, modwsrv checks the first data byte. If `FLUSHW` (defined in `'sys/stropts.h'`) is set, the write queue is flushed by use of the `flushq()` utility (see *STREAMS Utilities*). `flushq()` takes two arguments, the queue pointer and a flag. The flag indicates what should be flushed, data messages (`FLUSHDATA`) or everything (`FLUSHALL`). In the example, data includes `M_DATA`, `M_DELAY`, `M_PROTO`, and `M_PCPROTO` messages. The choice of what types of messages to flush is module specific.

Ordinary messages will be returned to the queue if

`canput(q->q_next)`

returns `'false'`, indicating the downstream path is blocked. The example continues with the remaining part of modwsrv processing `M_DATA` messages:

```

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

    if (!canput(q->q_next)) {
        putbq(q, mp);
        return;
    }
    /* Filter data, appending to queue */
    for (; mp != NULL; mp = next) {
        while (mp->b_rptr < mp->b_wptr) {
            if (*mp->b_rptr == '\n') {

```

```

        if (!bappend(&nbp, '\r'))
            goto push;
        if (!bappend(&nbp, *mp->b_rptr))
            goto push;
        mb->b_rptr++;
        continue;
    }
push:
    if (nbp)
        putnext(q, nbp);
    nbp = NULL;
    if (!canput(q->q_next)) {
        if (mp->b_rptr >= mp->b_wptr) {
            next = mp->b_cont;
            freeb(mp);
            mp = next;
        }
        if (mp)
            putbq(q, mp);
        return;
    }
} /* while */
next = mp->b_cont;
freeb(mp);
} /* for */
if (nbp)
    putnext(q, nbp);
} /* case M_DATA */
} /* switsh */
} /* while */
}

```

The differences in `M_DATA` processing between this and the example in Messages (see [7]4.5 " [8]Message Allocation and Freeing ") relate to the manner in which the new messages are forwarded and flow controlled. For the purpose of demonstrating alternative means of processing messages, this version creates individual new messages rather than a single message containing multiple message blocks. When a new message block is full, it is immediately forwarded with the `putnext(9)` routine rather than being linked into a single, large message (as was done in the Messages example). This alternative may not be desirable because message boundaries will be altered and because of the additional overhead of handling and scheduling multiple messages.

When the filter processing is performed (following push), flow control is checked [with `canput(9)`] after, rather than before, each new message is forwarded. This is done because there is no provision to hold the new message until the queue becomes unblocked. If the downstream path is blocked, the remaining part of the original message is returned to the queue. Otherwise, processing continues.

8.2 Module Flow Control

To utilize the *STREAMS* flow control mechanism, modules must use `service` procedures, invoke `canput(9)` before calling `putnext(9)`, and use appropriate values for the high and low water marks.

Module flow control limits the amount of data that can be placed on a queue. It prevents depletion of buffers in the buffer pool. Flow control is advisory in nature and it can be bypassed. It is managed by high and low water marks and regulated by `QWANTW` and `QFULL` flags. Module flow control is implemented by using the `canput(9)`, `getq(9)`, `putq(9)`, `putbq(9)`, `insq(9)`, and `rmvq(9)` routines.

The following scenario takes place normally in flow control when a module and driver are in sync:

- A driver sends data to a module using the `putnext(9)` routine, and the module's `put` procedure queues data using `putq()`. The `putq()` routine then increments the module's `q_count` by the number of bytes in the message and enables the `service` procedure. When *STREAMS* scheduling runs the `service` procedure, the `service` procedure then retrieves the data by calling the `getq(9)` utility, and `getq(9)` decrements `q_count` by an appropriate value.

If the module cannot process data at the rate at which the driver is sending the data, the following happens:

- The module's `q_count` goes above its high water mark, and the `QFULL` flag is set by `putq()`. The driver's `canput(9)` fails, and `canput(9)` sets `QWANTW` flag in the module's queue. The driver may send a command to the device to stop input, queue the data in its own queue, or drop the data. In the meanwhile, the module's `q_count` falls below its low water mark [by `getq(9)`] and `getq(9)` finds the nearest back queue with a `service` procedure and enables it. The scheduler then runs the `service` procedure.

The next two examples show a line discipline module's flow control. The first example is a read-side line discipline module:

```
/* read- side line discipline module flow control */

id_read_srv(q)
    queue_t *q;      /* pointer to read queue */
{
    mblk_t *mp;      /* original message */
    mblk_t *bp;      /* canonicalized message */

    while ((mp = getq(q)) != NULL) {
        switch (mp->b_datap->db_type) { /* type of message */
            case M_DATA:                /* data message */
                if (canput(q->q_next)) {
                    bp = read_canon(mp);
                    putnext(q, bp);
                } else {
                    putbq(q, mp); /* put message back in queue */
                    return;
                }
                break;
            default:
                if (mp->b_datap->db_type >= QPCTL)
                    putnext(q, mp); /* high priority message */
                else {                /* ordinary message */
                    if (canput(q->q_next))
                        putnext(q, mp);
                }
        }
    }
}
```

```

        else {
            putbq(q, mp);
            return;
        }
    }
    break;
}
}
}

```

The following shows a write-side line discipline module:

```

/* write-side line discipline module flow control */

id_write_srv(q)
    queue_t *q;      /* pointer to write queue */
{
    mblk_t *mp;      /* original message */
    mblk_t *bp;      /* canonicalized message */

    while ((mp = getq(q)) != NULL) {
        switch (mp->b_datap->db_type) { /* type of message */
            case M_DATA: /* data message */
                if (canput(q->q_next)) {
                    bp = write_canon(mp);
                    putnext(q, bp);
                } else {
                    putbq(q, mp);
                    return;
                }
                break;

            case M_IOCTL:
                ld_ioctl(q, mp);
                break;

            default:
                if (mp->b_datap->db_type >= QPCTL)
                    putnext(q, mp); /* high priority message */
                else { /* ordinary message */
                    if (canput(q->q_next))
                        putnext(q, mp);
                    else {
                        putbq(q, mp);
                        return;
                    }
                }
                break;
        }
    }
}

```

8.3 Module Design Guidelines

Module developers should follow these guidelines:

- Messages types that are not understood by the modules should be passed to the

nextmodule.

- The module that acts on an `M_IOCTL` message should send an `M_IOCACK` or `M_IOCNAK` message in response to the `ioctl`. If the module does not understand the `ioctl`, it should pass the `M_IOCTL` message to the next module.
- Modules should be designed in such way that they don't pertain to any particular driver but can be used by all drivers.
- In general, modules should not require the data in an `M_DATA` message to follow a particular format, such as a specific alignment. This makes it easier to arbitrarily push modules on top of each other in a sensible fashion. Not following this rule may limit module reusability.
- Filter modules pushed between a service user and a service provider may not alter the contents of the `M_PROTO` or `M_PCPROTO` block in messages. The contents of the data blocks may be manipulated, but the message boundaries must be preserved.

Also see [6]6.5 " [7]Design Guidelines " in [8]Chapter 6 [9]Overview of Modules and Drivers

9 STREAMS Drivers

9.1 Drivers

This chapter describes the operation of a *STREAMS* driver and also discusses some of the processing typically required in drivers.

Unlike a module, a device driver must have an interrupt routine so that it is accessible from a hardware interrupt as well as from the *Stream*. A driver can have multiple *Streams* connected to it. Multiple connections occur when more than one minor device of the same driver is in use and in the case of multiplexors (multiplexing is discussed in Multiplexing). However, these particular differences are not recognized by the *STREAMS* mechanism. They are handled by developer-provided code included in the driver procedures.

9.1.1 Overview of Drivers

This section provides a brief overview of the UNIX[®] system drivers. This is not an all-inclusive description, but an introduction and general information on drivers. UNIX[®] system drivers. This is not an all-inclusive description, but an introduction and general information on drivers.

A driver is software that provides an interface between the operating system and a device. The driver controls the device in response to kernel commands, and user-level programs access the device through system calls. The system calls interface with the file system and process control system, which in turn access the drivers. The driver provides and manages a path for the data to and from the hardware device, and services interrupts issued by the device controller.

9.1.1.1 Driver Classification

In general, drivers are grouped according to the type of the device they control, the access method (the way data are transferred), and the interface between the driver and the device.

The type can be hardware or software. A hardware driver controls a physical device such as a disk. A software driver, also called a pseudo device, controls software, which in turn may interface with a hardware device. The software driver may also support pseudo devices that have no associated physical device.

Drivers can be character-type or block-type, but many support both access methods. In character-type transfer, data are read a character at a time or as a variable length stream of bytes, the size of which is determined by the device. In block-type access, data transfer is performed on fixed-length blocks of data. Devices that support both block- and character-type access must have a separate special device file for each access method. Character access devices can also use "raw" (also called unbuffered) data transfer that takes place directly between user address space and the device. Unbuffered data transfer is used mainly for administrative functions where the speed of the specific operation is more important than overall system performance.

The driver interface refers to the system structures and kernel interfaces used by the driver. For example, *STREAMS* is an interface.

9.1.1.2 Driver Configuration

For a driver to be recognized as part of the system, information on driver type, where object code resides, interrupts, and so on, must be stored in appropriate files.

The following summarizes information needed to include a driver in the system (this information is unique to the AT&T 3B2):

`‘/etc/master.d’`

A master file supplies information to the system initialization software to describe different attributes of a driver. There is one master file for each driver in the system.

`‘/dev’`

This directory contains special files that provide applications with a way to access drivers via file operators.

9.1.1.3 Writing a Driver

All drivers are identified by a string of called the prefix. The prefix is defined in the master file for the driver and is added to the name of the driver routines. For example, the open routine for the driver with the "xyz" prefix is xyzopen.

Writing a driver differs from writing other C programs in the following ways:

- A driver does not have a main.c routine. Rather, driver entry points are given specific names and accessed through switch tables.
- A driver functions as a part of the kernel. Consequently, a poorly written driver can degrade system performance or corrupt the system.
- A driver cannot use system calls or the C library, because the driver functions at a lower level.
- A driver cannot use floating point arithmetic. A driver cannot use archives or shared libraries,
- but frequently used subroutines can be put in separate files in the source code directory for the driver.

Driver code, like other system software, uses the advanced C language capabilities. These include: bit manipulation capabilities, casting of data types, and use of header files for defining and declaring global data structures.

Driver code includes a set of entry point routines:

- initialization entry points that are accessed through arrays during system initialization.
- switch table entry points that are accessed through bdevsw (block- access) and cdevsw (character-access) switch tables when the appropriate system call is issued.

The following lists rules of driver development:

- All drivers must have an associated file in the `‘master.d’` directory.
- All drivers should have system header files that define data structures used in the driver.
- Drivers may have an init and/or astart routine to initialize the driver. Software drivers will usually have little to initialize, because there is no hardware involved. An init

routine is used when a driver needs to initialize but does not need any system services. `init` routines are run before system services are initialized. When a driver needs to do initialization that requires system services, a `start` routine is used. The `start` routines are run after system services have been initialized.

- Drivers will have `open` and `close` routines. Most drivers will have an interrupt handler routine. The driver developer is responsible for supplying an interrupt routine for the device's driver. The *UNIX* system provides a few interrupt handling routines for hardware interrupts, but the developer has to supply the specifics about the device. In general, a prefixint interrupt routine should be written for any device that does not send separate transmit and receive interrupts. *TTY* devices that request separate transmit and receive interrupts can have two separate interrupt routines associated with them; `prefixxinit` to transmit an interrupt, and `prefixrint` to receive an interrupt.
- Most drivers will have static subordinate driver routines to provide the functionality for the specific device. The names of these routines should include the driver prefix, although this is not absolutely required since the routine is declared as static.
- A bootable object file and special device files are also needed for a driver to be fully functional.

9.1.1.4 Major and Minor Device Numbers

The *UNIX System V* operating system identifies and accesses peripheral devices by major and minor numbers. When a driver is installed and a special device file is created, a device then appears to the user application as a file. A device is accessed by opening, reading, writing, and closing a special device file that has the proper major and minor device numbers.

The major number identifies a driver for a controller. The minor number identifies a specific device. Major numbers are assigned sequentially by either the system initialization software at boot time for hardware devices, by a program such as `drvinstall`, or by administrator direction. The major number for a software device is assigned automatically by the `drvinstall` command. Minor numbers are designated by the driver developer.

Major and minor numbers can be external or internal.

External major numbers are those visible to the user.

Internal major numbers serve as an index into the `cdevsw` and `bdevsw` switch tables. These are assigned by the autoconfiguration process when drivers are loaded and they may change every time a full configuration boot is done. The system uses the *MAJOR* table to translate external major numbers to the internal major numbers needed to access the switch tables.

One driver may control several devices, but each device will have its own external major number and all those external major numbers are mapped to one internal major number for the driver.

Minor numbers are determined differently for different types of devices. Typically, minor numbers are an encoding of information needed by the controller board.

External minor numbers are controlled by a driver developer, although there are conventions enforced for some types of devices by some utilities. For example, a tape drive may interface

with a hardware controller (device) to which several tape drives (subdevices) are attached. All tape drives attached to one controller will have the same external major number, but each drive will have a different external minor number.

The *MAJOR* and *MINOR* tables map external major and minor numbers to the internal major number. The switch tables will have only as many entries as required to support the drivers installed on the system. Switch table entry points are activated by system calls that reference a special device file that supplies the external major number and instructions on whether to use *bdevsw* or *cdevsw*. By mapping the external major number to the corresponding internal major number in the *MAJOR* table, the system knows which driver routine to activate. The routines *getmajor()* and *getminor()* return an internal major and minor number for the device. The routines *getemajor()* and *geteminor()* return an external major and minor number for the device.

9.1.2 STREAMS Drivers

At the interface to hardware devices, character I/O drivers have interrupt entry points; at the system interface, those same drivers generally have direct entry points (routines) to process *open*, *close*, *read*, *write*, *poll*, and *ioctl* system calls.

STREAMS device drivers have interrupt entry points at the hardware device interface and have direct entry points only for the *open* and *close* system calls. These entry points are accessed via *STREAMS*, and the call formats differ from traditional character device drivers. (*STREAMS* drivers are character drivers, too. We call the non-*STREAMS* character drivers traditional character drivers or non-*STREAMS* character drivers.) The *put* procedure is a driver's third entry point, but it is a message (not system) interface. The *Stream head* translates *write* and *ioctl* calls into messages and sends them downstream to be processed by the driver's write queue *put* procedure. *read* is seen directly only by the *Stream head*, which contains the functions required to process system calls. A driver does not know about system interfaces other than *open* and *close*, but it can detect the absence of a *read* indirectly if flow control propagates from the *Stream head* to the driver and affects the driver's ability to send messages upstream.

For input processing, when the driver is ready to send data or other information to a user process, it does not wake up the process. It prepares a message and sends it to the read queue of the appropriate (minor device) *Stream*. The driver's *open* routine generally stores the queue address corresponding to this *Stream*.

For output processing, the driver receives messages in place of a *write* call. If the message can not be sent immediately to the hardware, it may be stored on the driver's write message queue. Subsequent output interrupts can remove messages from this queue.

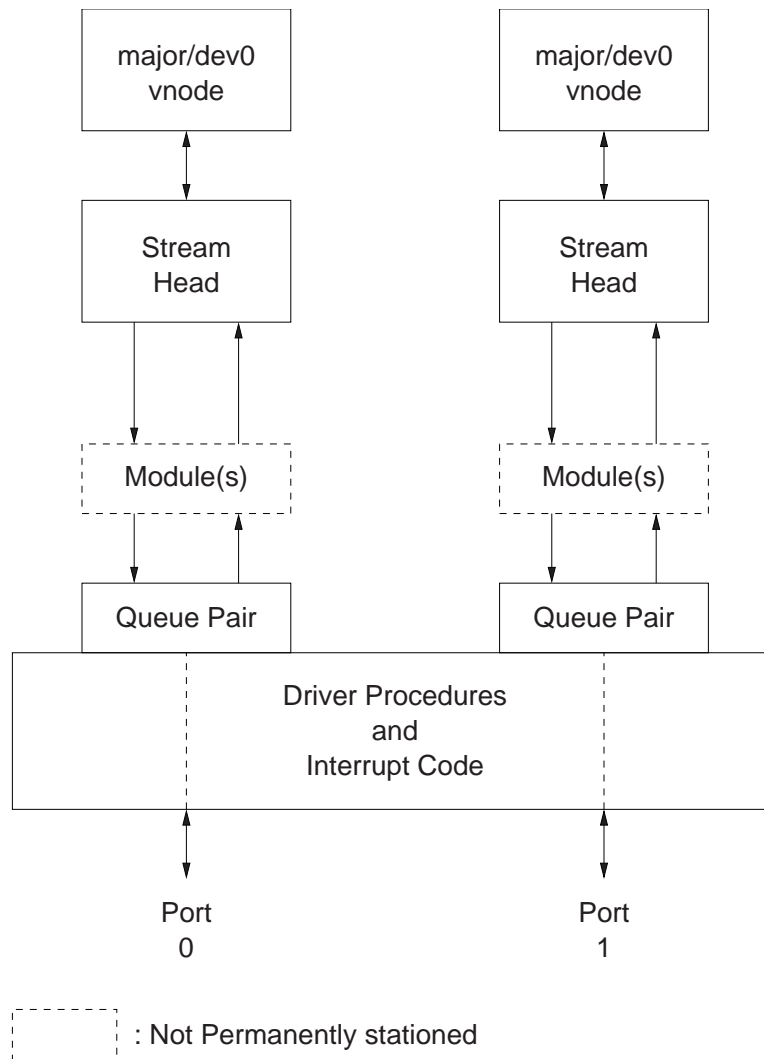
Figure 9.1 shows multiple *Streams* (corresponding to minor devices) to a common driver. There are two distinct *Streams* opened from the same major device. Consequently, they have the same *streamtab* and the same driver procedures.

The configuration mechanism distinguishes between *STREAMS* devices and traditional character devices, because system calls to *STREAMS* drivers are processed by *STREAMS* routines, not by the *UNIX* system driver routines. In the *cdevsw* file, the field *d_str* provides this distinction. See [Appendix E \[STREAMS Configuration\]](#), page 323, for details.

Multiple instantiations (minor devices) of the same driver are handled during the initial open for each device. Typically, the queue address is stored in a driver-private structure array indexed by the minor device number. This is for use by the interrupt routine which needs to translate from device number to a particular *Stream*. The `q_ptr` of the queue will point to the private data structure entry. When the messages are received by the queue, the calls to the driver `put` and `service` procedures pass the address of the queue, allowing the procedures to determine the associated device.

A driver is at the end of a *Stream*. As a result, drivers must include standard processing for certain message types that a module might simply be able to pass to the next component.

During the open and close routine the kernel locks the device snode. Thus only one open or close can be active at a time per major/minor device pair.

Figure 9.1: *Device Driver Streams*

9.1.2.1 Printer Driver Example

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

Open handling

A driver is passed a device number or is asked to select one.

Flush handling

A driver must loop `M_FLUSH` messages back upstream.

ioctl handling

A driver must send a negative acknowledgement for `ioctl` messages it does not understand. This is discussed under [8]6.2 " [9]Module and Driver `ioctls` " in Overview of Modules and Drivers.

Declarations

The driver declarations are as follows (see also "Module and Driver Declarations" in [10]Chapter 6 [11]Overview of Modules and Drivers):

```
/* Simple line printer driver */

#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/dir.h>           /* required for user.h */
#include <sys/signal.h>        /* required for user.h */
#include <sys/user.h>
#include <sys/errno.h>
#include <sys/cred.h>
#include <sys/ddi.h>

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

static int lopen(), lpclose(), lpwput();

static struct qinit rinit = {
    NULL, NULL, lopen, lpclose, NULL, &minfo, NULL
};

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

struct streamtab lpinfo = { &rinit, &winit, NULL, NULL };

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

/* This is a private data structure, one per minor device number. */

struct lp {
    short flags;           /* flags -- see below */
    mblk_t *msg;           /* current message being output */
    queue_t *qptr;         /* back pointer to write queue */
};

/* Flags bits */
#define BUSY 1              /* device is running and interrupt is
                             pending */

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

```
int lpdevflag = 0;
```

Configuring a *STREAMS* driver requires only the `streamtab` structure to be externally accessible. All other *STREAMS* driver procedures would typically be declared static.

The `streamtab` structure must be defined as `prefixinfo`, where `prefix` is the value of the `prefix` field in the master file for this driver. The values in the module name and ID fields in the `module_info` structure should be unique in the system. Note that, as in character I/O drivers, extern variables are assigned values in the master file when configuring drivers or modules.

There is no read-side `put` or `service` procedure. The flow control limits for use on the write-side are 50 bytes for the low water mark and 150 bytes for the high water mark. The private `lp` structure is indexed by the minor device number and contains these elements:

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

Driver Open

The *STREAMS* mechanism allows only one *Stream* per minor device. The driver open routine is called whenever a *STREAMS* device is opened. Opening also allocates a private data structure. The driver open, `lpopen` in this example, has the same interface as the module open:

```
static int
lpopen(q, devp, flag, sflag, credp)
    queue_t *q;      /* read queue */
    dev_t *devp;
    int flag;
    int sflag;
    cred_t *credp;
{
    struct lp *lp;
    dev_t device;

    if (sflag)                /* check if non-driver open */
        return ENXIO;

    device = getminor(*devp);
    if (device >= lp_cnt)
        return ENXIO;

    if (q->q_ptr)              /* Check if open already.  q_ptr is
                                assigned below */
        return EBUSY;

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

```

    return 0;
}

```

The *Stream* flag, `sflag`, must have the value 0, indicating a normal driver open. `devp` is a pointer to the major/minor device number for this port. After checking `sflag`, the *STREAMS* open flag, `lopen` extracts the minor device pointed to by `devp`, using the `getminor()` function. `credp` is a pointer to a credentials structure.

The minor device number selects a printer. The device number pointed to by `devp` must be less than `lp_cnt`, the number of configured printers. Otherwise failure occurs.

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

There are no physical pointers between queues. `WR` is a queue pointer macro. `WR(q)` generates the write pointer from the read pointer. `RD(9)` and `OTHER(9)` are also the queue pointer macros. `RD(q)` generates the read pointer from the write pointer, and `OTHER(9)` generates the mate pointer from either. With the *DDI*, `WR(9)`, `RD(9)`, and `OTHER(9)` are functions.

Driver Flush Handling

The following write `put` procedure, `lpwput`, illustrates driver `M_FLUSH` handling. NOTE that all drivers are expected to incorporate flush handling.

If `FLUSHW` is set, the write message queue is flushed, and (in this example) the leading message (`lp->msg`) is also flushed. `spl5` is used to protect the critical code, assuming the device interrupts at level 5.

Normally, if `FLUSHR` is set, the read queue would be flushed. However, in this example, no messages are ever placed on the read queue, so it is not necessary to flush it. The `FLUSHW` bit is cleared and the message is sent upstream using `qreply()`. If `FLUSHR` is not set, the message is discarded.

The *Stream head* always performs the following actions on flush requests received on the read-side from downstream. If `FLUSHR` is set, messages waiting to be sent to user space are flushed. If `FLUSHW` is set, the *Stream head* clears the `FLUSHR` bit and sends the `M_FLUSH` message downstream. In this manner, a single `M_FLUSH` message sent from the driver can reach all queues in a *Stream*. A module must send two `M_FLUSH` messages to have the same affect.

`lpwput` enqueues `M_DATA` and `M_IOCTL` messages and, if the device is not busy, starts output by calling `lpout`. Messages types that are not recognized are discarded.

```

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

```

```

lp = (struct lp *) q->q_ptr;
switch (mp->b_datap->db_type) {

default:
    freemsg(mp);
    break;

case M_FLUSH:                /* Canonical flush handling */
    if (*mp->b_rptr & FLUSHW) {
        flushq(q, FLUSHDATA);
        s = sp15();           /* also flush lp->msg since it is
                                logically * at the head of the
                                write queue */
        if (lp->msg) {
            freemsg(lp->msg);
            lp->msg = NULL;
        }
        splx(s);
    }
    if (*mp->b_rptr & FLUSHR) {
        *mp->b_rptr &= ~FLUSHW;
        greply(q, mp);
    } else
        freemsg(mp);
    break;

case M_IOCTL:

case M_DATA:
    putq(q, mp);
    s = sp15();
    if (!(lp->flags & BUSY))
        lpout(lp);
    splx(s);
}
}

```

Driver Interrupt

The following example shows the interrupt routine in the printer driver.

lpint is the driver interrupt handler routine.

lpout simply takes a character from the queue and sends it to the printer. For convenience, the message currently being output is stored in lp->msg.

lpoutchar sends a character to the printer and interrupts when complete. Printer interface options need to be set before being able to print.

```

/* Device interrupt routine */

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

    lp = &lp_lp[device];

```



```

        if (!(lp->flags & BUSY)) {
            cmn_err(CE_WARN, "~lp: unexpected interrupt\n");
            return;
        }
        lp->flag &= ~BUSY;
        lpout(lp);
    }

    /* Start output to device - used by put procedure and driver */
    lpout(lp)
    {
        register struct lp *lp;

        register mblk_t *bp;
        queue_t *q;

        q = lp->qptr;

    loop:
        if ((bp = lp->msg) == NULL) {           /* no current message */
            if ((bp = getq(q)) == NULL) {
                lp->flags &= NBUSY;
                return;
            }
            if (bp->b_datap->db_type == M_IOCTL) {
                lpdoioctl(lp, bp);
                goto loop;
            }
            lp->msg = bp;                       /* new message */
        }

        if (bp->b_rptr >= bp->b_wptr) {           /* validate message */
            bp = lp->msg->b_cont;
            lp->msg->b_cont = NULL;
            freeb(lp->msg);
            lp->msg = bp;
            goto loop;
        }
        lpoutchar(lp, *bp->b_rptr++);           /* output one character */
        lp->flags |= BUSY;
    }
}

```

Driver Close

The driver close routine is called by the *Stream head*. Any messages left on the queue will be automatically removed by *STREAMS*. The *Stream* is dismantled and the data structures are de-allocated.

```

static int
lpclose(q, flag, credp)
    queue_t *q;      /* read queue */
    int flag;
    cred_t *credp;
{
    struct lp *lp;
    int s;

```

```

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

    /* Free message, queue is automatically flushed by STREAMS */

    s = sp15();
    if (lp->msg) {
        freemsg(lp->msg);
        lp->msg = NULL;
    }
    splx(s);
    lp->flags = 0;
}

```

9.1.2.2 Driver Flow Control

The same utilities (described in Modules) and mechanisms used for module flow control are used by drivers.

When the message is queued, `putq()` increments the value of `q_count` by the size of the message and compares the result against the driver's write high water limit (`q_hiwat`) value. If the count exceeds `q_hiwat`, the `putq()` utility routine will set the internal `FULL` indicator for the driver write queue. This will cause messages from upstream to be halted [`canput(9)` returns `FALSE`] until the write queue count reaches `q_lowat`. The driver messages waiting to be output are dequeued by the driver output interrupt routine with `getq(9)`, which decrements the count. If the resulting count is below `q_lowat`, the `getq(9)` routine will back-enable any upstream queue that had been blocked.

Device drivers typically discard input when unable to send it to a user process. However, *STREAMS* allows flow control to be used on the driver read-side to handle temporary upstream blocks.

To some extent, a driver or a module can control when its upstream transmission will become blocked. Control is available through the `M_SETOPTS` message (see [Section B.1 \[Message Types\]](#), page 281) to modify the *Stream head* read-side flow control limits.

9.2 Cloning

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

The clone device may be useful, for example, in a networking environment where a protocol pseudo device driver requires each user to open a separate *Stream* over which it will establish communication.

The decision to implement a *STREAMS* driver as a cloneable device is made by the designers of the device driver.

Knowledge of clone driver implementation is not required to use it. A description is presented here for completeness and to assist developers who must implement their own clone driver.

There are two ways to create a clone device node in the file system. The first is to have a node with major number 63 (major of the clone driver) and with a minor number equal to the major number of the real device one wants to open. For example, `/dev/starlan00` might be major 40, minor 0 (normal open), and `/dev/starlan` might be major 63, minor 40 (clone open).

The second way to create a clone device node is for the driver to designate a special minor device as its clone entry point. Here, `/dev/starlan` might be major 40, minor 0 (clone open).

The former example will cause `sflag` to be set to `CLONEOPEN` in the open routine when `/dev/starlan` is opened. The latter will not. Instead, in the latter case the driver has decided to designate a special minor device as its clone interface. When the clone is opened, the driver knows that it should look for an unused minor device. This implies that the reserved minor for the clone entry point will never be given out.

In either case, the driver returns the new device number as:

```
*devp = makedevice(getmajor(*devp), newminor)
```

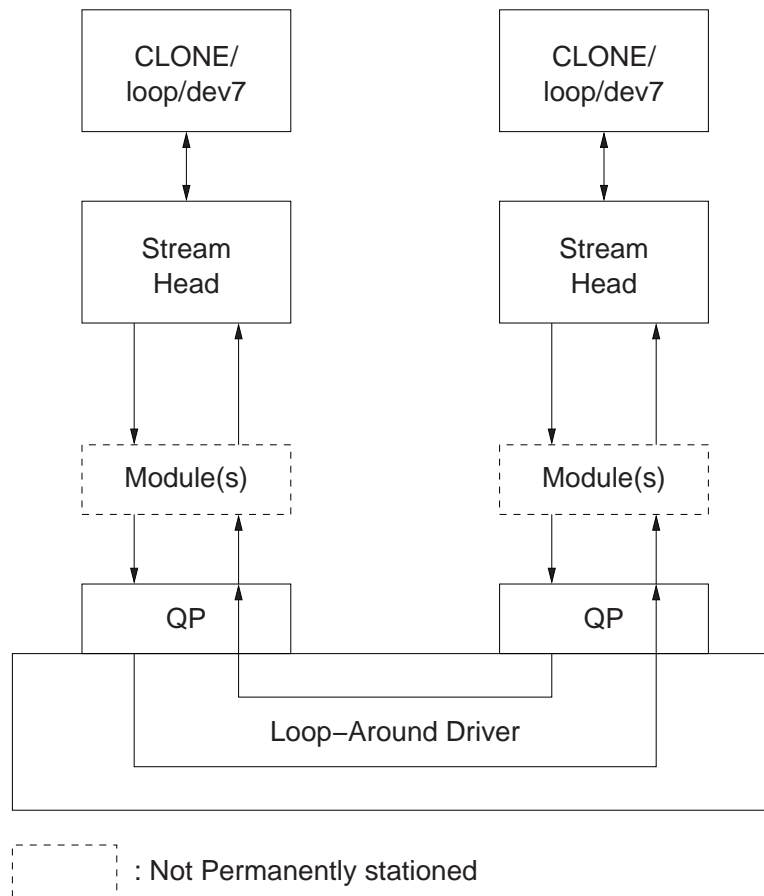
`makedevice` is unique to the *DDI* interface. If the *DDI* interface is not used, `makedev` can be used instead of `makedevice`.

9.3 Loop-Around Driver

The loop-around driver is a pseudo driver that loops data from one open *Stream* to another open *Stream*. The user processes see the associated files almost like a full-duplex pipe. The *Streams* are not physically linked. The driver is a simple multiplexor that passes messages from one *Stream*'s write queue to the other *Stream*'s read queue.

To create a connection, a process opens two *Streams*, obtains the minor device number associated with one of the returned file descriptors, and sends the device number in an `I_STR ioctl(2)` to the other *Stream*. For each open, the driver open places the passed queue pointer in a driver interconnection table, indexed by the device number. When the driver later receives the `I_STR` as an `M_IOCTL` message, it uses the device number to locate the other *Stream*'s interconnection table entry, and stores the appropriate queue pointers in both of the *Streams*' interconnection table entries.

Subsequently, when messages other than `M_IOCTL` or `M_FLUSH` are received by the driver on either *Stream*'s write-side, the messages are switched to the read queue following the driver on the other *Stream*'s read-side. The resultant logical connection is shown in [Figure 9.2](#) (in the figure, the abbreviation QP represents a queue pair). Flow control between the two *Streams* must be handled by special code since *STREAMS* will not automatically propagate flow control information between two *Streams* that are not physically interconnected.

Figure 9.2: *Loop-Around Streams*

The next example shows the loop-around driver code. A master file to configure the loop driver is shown in [Appendix E \[STREAMS Configuration\]](#), page 323. The loop structure contains the interconnection information for a pair of *Streams*. `loop_loop` is indexed by the minor device number. When a *Stream* is opened to the driver, the address of the corresponding `loop_loop` element is placed in `q_ptr` (private data structure pointer) of the read-side and write-side queues. Since *STREAMS* clears `q_ptr` when the queue is allocated, a 'NULL' value of `q_ptr` indicates an initial open. `loop_loop` is used to verify that this *Stream* is connected to another open *Stream*.

The declarations for the driver are:

```

/* Loop-around driver */

#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/stream.h>
#include <sys/stropts.h>

```

```

#include <sys/dir.h>
#include <sys/signal.h>
#include <sys/user.h>
#include <sys/errno.h>
#include <sys/cred.h>
#include <sys/ddi.h>

static struct module_info minfo = {
    0xee12, "loop", 0, INFP SZ, 512, 128
};

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

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

struct streamtab loopinfo = { &rinit, &winit, NULL, NULL };

struct loop {
    queue_t *qptr;      /* back pointer to write queue */
    queue_t *oqptr;     /* pointer to connected read queue */
}

#define LOOP_SET (('1' << 8 | 1) /* should be in a . h file */
extern struct loop loop_loop[];
extern int loop_cnt;
int loopdevflag = 0;

```

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

```

static int
loopopen(q, devp, flag, sflag, credp)
    queue_t *q;
    dev_t *devp;
    int flag;
    int sflag;
    cred_t *credp;
{
    struct loop *loop;

    dev_t newminor;

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

```

```

        newminor = getminor(*devp);

    if (newminor >= loop_cnt)
        return ENXIO;
    /* construct new device number and reset devp */

    /* getmajor gets the external major number, if (sflag ==
       CLONEOPEN) */

    if (q->q_ptr)                /* already open */
        return 0;

    *devp = makedev(getemajor(*devp), newminor);
    loop = &loop_loop[newminor];
    WR(q)->q_ptr = (char *) loop;
    q->q_ptr = (char *) loop;
    loop->qptr = WR(q);
    loop->oqptr = NULL;

    return 0;
}

```

In `loopopen`, `sflag` can be `CLONEOPEN`, indicating that the driver should pick an unused minor device (i.e., the user does not care which minor device is used). In this case, the driver scans its private `loop_loop` data structure to find an unused minor device number. If `sflag` has not been set to `CLONEOPEN`, the passed-in minor device specified by `getminor->(*devp)` is used. Since the messages are switched to the read queue following the other *Stream*'s read-side, the driver needs a `put` procedure only on its write-side:

```

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

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

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

        iocp = (struct iocblk *) mp->b_rptr;
        switch (iocp->ioc_cmd) {
        case LOOP_SET:
        {
            int to          /* other minor device */
            /*
             * Sanity check.  ioc_count contains the amount of
             * user supplied data which must equal the size of an int.
             */
            if (iocp->ioc_count != sizeof(int)) {
                error = EINVAL;
                goto iocnak;
            }

```

```

    }
    /* fetch other dev from 2nd message block */
    to = *(int *) mp->b_cont->b_rptr;
    /*
     * More sanity checks. The minor must be in range, open already.
     * Also, this device and the other one must be disconnected.
     */

    if (to >= loop_cnt || to < 0 || !loop_loop[to].qptra) {
        error = ENXIO;
        goto iocnak;
    }
    if (loop->qptra || loop_loop[to].qptra) {
        error = EBUSY;
        goto iocnak;
    }
    /* Cross connect streams via the loop structures */
    loop->qptra = RD(loop_loop[to].qptra);
    loop_loop[to].qptra = RD(q);
    /*
     * Return successful ioctl. Set ioc_count
     * to zero, since no data are returned.
     */

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

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

}
break;
}

```

loopwput shows another use of an `I_STR` `ioctl` call (see [8]Chapter 6 [9]Overview of Modules and Drivers , "Module and Driver ioctls"). The driver supports a `LOOP_SET` value of `ioc_cmd` in the `iocblk` of the `M_IOCTL` message. `LOOP_SET` instructs the driver to connect the current open *Stream* to the *Stream* indicated in the message. The second block of the `M_IOCTL` message holds an integer that specifies the minor device number of the *Stream* to connect to.

The driver performs several sanity checks: Does the second block have the proper amount of data? Is the "to" device in range? Is the "to" device open? Is the current *Stream* disconnected? Is the "to" *Stream* disconnected?

If everything checks out, the read queue pointers for the two *Streams* are stored in the respective `oqptr` fields. This cross-connects the two *Streams* indirectly, via `loop_loop`.

Canonical flush handling is incorporated in the `put` procedure:

```

case M_FLUSH:
    if (*mp->b_rptr & FLUSHW) {
        flushq(q, FLUSHALL);          /* write */
        flushq(loop->oqptr, FLUSHALL);
        /* read on other side equals write on this side */
    }
    if (*mp->b_rptr & FLUSHR) {
        flushq(RD(q), FLUSHALL);
        flushq(WR(loop->oqptr), FLUSHALL);
    }
    switch (*mp->b_rptr) {

    case FLUSHW:
        *mp->b_rptr = FLUSHR;
        break;

    case FLUSHR:
        *mp->b_rptr = FLUSHW;
        break;
    }
    putnext(loop->oqptr, mp);
    break;

default:
    /* If this Stream isn't connected, send M_ERROR upstream. */
    if (loop->oqptr == NULL) {
        freemsg(mp);
        putctl1(RD(q)->q_next, M_ERROR, ENXIO);
        break;
    }
    putq(q, mp);
}
}

```

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

Certain message types can be sent upstream by drivers and modules to the *Stream head* where they are translated into actions detectable by user process(es). The messages may also modify the state of the *Stream head*:

M_ERROR Causes the *Stream head* to lock up. Message transmission between *Stream* and user processes is terminated. All subsequent system calls except `close(2)` and `poll(2)` will fail. Also causes an `M_FLUSH` clearing all message queues to be sent downstream by the *Stream head*.

M_HANGUP Terminates input from a user process to the *Stream*. All subsequent system calls that would send messages downstream will fail. Once the *Stream head* read message queue is empty, *EOF* is returned on reads. Can also result in the {SIGHUP} signal being sent to the process group.

M_SIG

M_PCSIG Causes a specified signal to be sent to a process.

putctl1() and putctl() are utilities that allocate a non-data (i.e., not M_DATA, M_DELAY, M_PROTO, or M_PCPROTO) type message, place one byte in the message (for putctl1) and call the put procedure of the specified queue.

Service procedures are required in this example on both the write-side and read-side for flow control:

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

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

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

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

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

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

static int
looprsrv(q)
    queue_t *q;
{
    /* Enter only when "back enabled" by flow control */

    struct loop *loop;

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

    /* manually enable write service procedure */

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

```
    }
```

The write **service** procedure, `loopwsvr`, takes on the canonical form. The queue being written to is not downstream, but upstream (found via `oqptr`) on the other *Stream*.

In this case, there is no read-side **put** procedure so the read **service** procedure, `looprsrv`, is not scheduled by an associated **put** procedure, as has been done previously. `looprsrv` is scheduled only by being back enabled when its upstream becomes unstuck from flow control blockage. The purpose of the procedure is to re-enable the writer (`loopwsvr`) by using `oqptr` to find the related queue. `loopwsvr` can not be directly back-enabled by *STREAMS* because there is no direct queue linkage between the two *Streams*. Note that no message ever gets queued to the read **service** procedure. Messages are kept on the write-side so that flow control can propagate up to the *Stream head*. The `qenable()` routine schedules the write-side **service** procedure of the other *Stream*.

`loopclose` breaks the connection between the *Streams*:

```
static int
loopclose(q, flag, credp)
    queue_t *q;
    int flag;
    cred_t *credp;
{
    register struct loop *loop;

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

    /* If we are connected to another stream, break the linkage, and
       send a hangup message. The hangup message causes the stream
       head to fail writes, allow the queued data to be read
       completely, and then return EOF on subsequent reads. */
    if (loop->oqptr) {
        ((struct loop *) loop->oqptr->q_ptr)->oqptr = NULL;
        putctl(loop->oqptr->q_next, M_HANGUP);
        loop->oqptr = NULL;
    }
}
```

`loopclose` sends an `M_HANGUP` message up the connected *Stream* to the *Stream head*.

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

9.4 Driver Design Guidelines

Driver developers should follow these guidelines:

- Messages that are not understood by the drivers should be freed.
- A driver must process an `M_IOCTL` message. Otherwise, the *Stream head* will block for an `M_IOCNAK` or `M_IOCACK` until the timeout (potentially infinite) expires.
- If a driver does not understand an `ioctl`, an `M_IOCNAK` message must be sent to upstream.
- Terminal drivers must always acknowledge the *EUC* `ioctls` whether they understand them or not.

- The *Stream head* locks up the *Stream* when it receives an `M_ERROR` message, so driver developers should be careful when using the `M_ERROR` message.
- If a driver wants to allocate a controlling terminal, it should send an `M_SETOPTS` message with the `SO_ISTTY` flag set upstream.
- A driver must be a part of the kernel for it to be opened.

Also see [6]6.5 " [7]Design Guidelines " in [8]Chapter 6 [9]Overview of Modules and Drivers .

10 STREAMS Multiplexing

10.1 Multiplexing

This chapter describes how *STREAMS* multiplexing configurations are created and also discusses multiplexing drivers. A *STREAMS* multiplexor is a driver with multiple *Streams* connected to it. The primary function of the multiplexing driver is to switch messages among the connected *Streams*. Multiplexor configurations are created from user level by system calls.

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

Multiple *Streams* can be connected above a driver by use of `open(2)` calls. This was done for the loop around driver and for the driver handling multiple minor devices in Drivers. There is no difference between the connections to these drivers, only the functions performed by the driver are different. In the multiplexing case, the driver routes data between multiple *Streams*. In the device driver case, the driver routes data between user processes and associated physical ports. Multiplexing with *Streams* connected above is referred to as an N-to-1, or upper, multiplexor. *STREAMS* does not provide any facilities beyond `open(2)` and `close(2)` to connect or disconnect upper *Streams* for multiplexing purposes.

From the driver's perspective, upper and lower configurations differ only in the way they are initially connected to the driver. The implementation requirements are the same: route the data and handle flow control. All multiplexor drivers require special developer-provided software to perform the multiplexing data routing and to handle flow control. *STREAMS* does not directly support flow control among multiplexed *Streams*.

M-to-N multiplexing configurations are implemented by using both of the above mechanisms in a driver.

As discussed in Drivers, the multiple *Streams* that represent minor devices are actually distinct *Streams* in which the driver keeps track of each *Stream* attached to it. The *STREAMS* subsystem does not recognize any relationship between the *Streams*. The same is true for *STREAMS* multiplexors of any configuration. The multiplexed *Streams* are distinct and the driver must be implemented to do most of the work.

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

10.1.1 Building a Multiplexor

This section builds a protocol multiplexor with the multiplexing configuration shown in Figure 10.1. To free users from the need to know about the underlying protocol structure, a user-level daemon process will be built to maintain the multiplexing configuration. Users can then access the transport protocol directly by opening the transport protocol (TP) driver device node.

An internetworking protocol driver (IP) routes data from a single upper *Stream* to one of two lower *Streams*. This driver supports two *STREAMS* connection beneath it. These connections are to two distinct networks; one for the *IEEE 802.3* standard via the 802.3 driver, and other to the *IEEE 802.4* standard via the 802.4 driver. The TP driver multiplexes upper *Streams* over a single *Stream* to the IP driver.

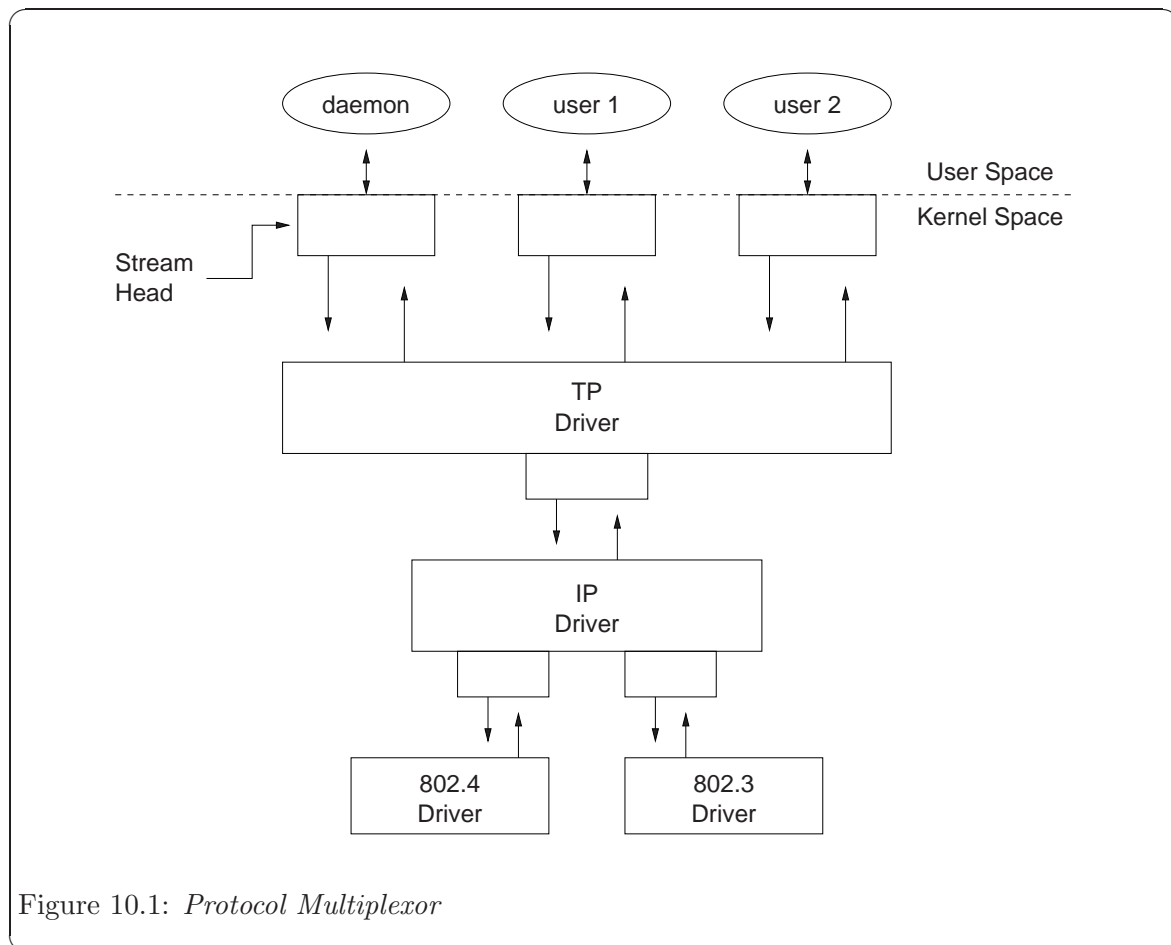


Figure 10.1: *Protocol Multiplexor*

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

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/stat.h>
```

```

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

main()
{
    int fd_802_4, fd_802_3, fd_ip, fd_tp;

    /* daemon-ize this process */

    switch (fork()) {
    case 0:
        break;
        case -1;
        perror("fork failed");
        exit(2);
    default:
        exit(0);
    }
    setsid();
}

```

This multi-level multiplexed *Stream* configuration will be built from the bottom up. Therefore, the example begins by first constructing the Internet Protocol (IP) multiplexor. This multiplexing device driver is treated like any other software driver. It owns a node in the *UNIX* file system and is opened just like any other *STREAMS* device driver.

The first step is to open the multiplexing driver and the 802.4 driver, thus creating separate *Streams* above each driver as shown in Figure 10.2. The *Stream* to the 802.4 driver may now be connected below the multiplexing IP driver using the `I_LINK` `ioctl` call.

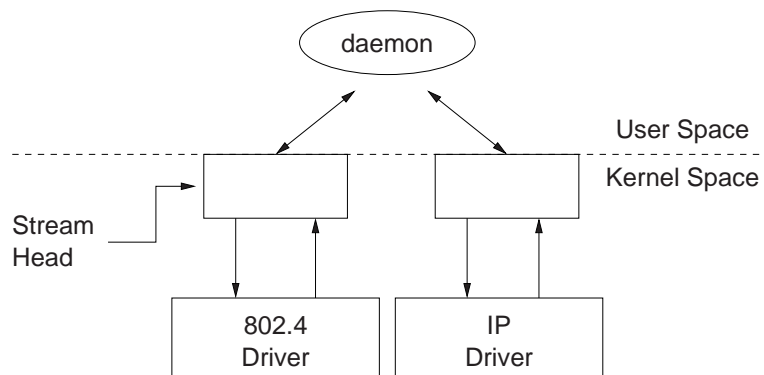


Figure 10.2: *Before Link*

The sequence of instructions to this point is:

```

if ((fd_802_4 = open("/dev/802_4", O_RDWR)) < 0) {
    perror("open of /dev/802_4 failed");
    exit(1);
}

```

```

if ((fd_ip = open("/dev/ip", 0_RDWR)) < 0) {
    perror("open of /dev/ip failed");
    exit(2);
}

/* now link 802.4 to underside of IP */

if (ioctl(fd_ip, I_LINK, fd_802_4) < 0) {
    perror("I_LINK ioctl failed");
    exit(3);
}

```

I_LINK takes two file descriptors as arguments. The first file descriptor, `fd_ip`, must reference the *Stream* connected to the multiplexing driver, and the second file descriptor, `fd_802_4`, must reference the *Stream* to be connected below the multiplexor. Figure 10.3 shows the state of these *Streams* following the I_LINK call. The complete *Stream* to the 802.4 driver has been connected below the IP driver. The *Stream head's* queues of the 802.4 driver will be used by the IP driver to manage the lower half of the multiplexor.

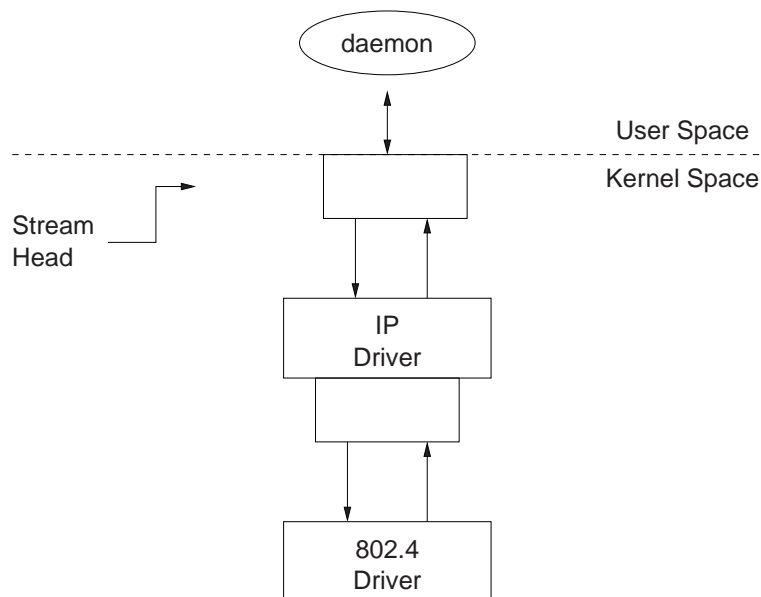


Figure 10.3: *IP Multiplexor After First Link*

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

The following sequence of system calls is used to continue building the internetworking protocol multiplexor (IP):


```

if ((fd_802_3 = open("/dev/802_3", O_RDWR)) < 0) {
    perror("open of /dev/802_3 failed");
    exit(4);
}

if (ioctl(fd_ip, I_LINK, fd_802_3) < 0) {
    perror("I_LINK ioctl failed");
    exit(5);
}

```

All links below the IP driver have now been established, giving the configuration in [Figure 10.4](#).

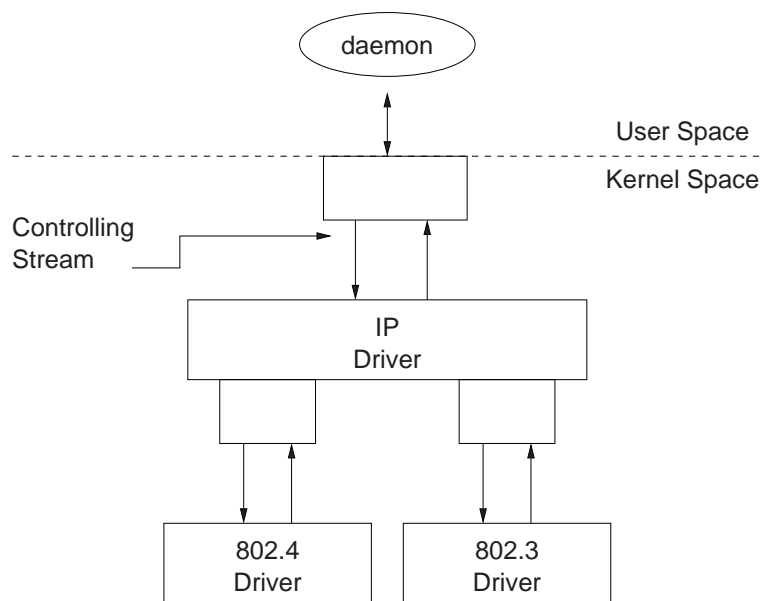


Figure 10.4: *IP Multiplexor*

The *Stream* above the multiplexing driver used to establish the lower connections is the controlling *Stream* and has special significance when dismantling the multiplexing configuration. This will be illustrated later in this chapter. The *Stream* referenced by `fd_ip` is the controlling *Stream* for the IP multiplexor.

The order in which the *Streams* in the multiplexing configuration are opened is unimportant. If it is necessary to have intermediate modules in the *Stream* between the IP driver and media drivers, these modules must be added to the *Streams* associated with the media drivers (using `I_PUSH`) before the media drivers are attached below the multiplexor.

The number of *Streams* that can be linked to a multiplexor is restricted by the design of the particular multiplexor. The manual page describing each driver describes such restrictions. However, only one `I_LINK` operation is allowed for each lower *Stream*; a single *Stream* cannot be linked below two multiplexors simultaneously.

Continuing with the example, the IP driver will now be linked below the transport protocol (TP) multiplexing driver. As seen earlier in [Figure 10.1](#), only one link will be supported below the transport driver. This link is formed by the following sequence of system calls:

```
if ((fd_tp = open("/dev/tp", 0_RDWR)) < 0) {
    perror("open of /dev/tp failed");
    exit(6);
}

if (ioctl(fd_tp, I_LINK, fd_ip) < 0) {
    perror("I_LINK ioctl failed");
    exit(7);
}
```

The multi-level multiplexing configuration shown in [Figure 10.5](#) has now been created.

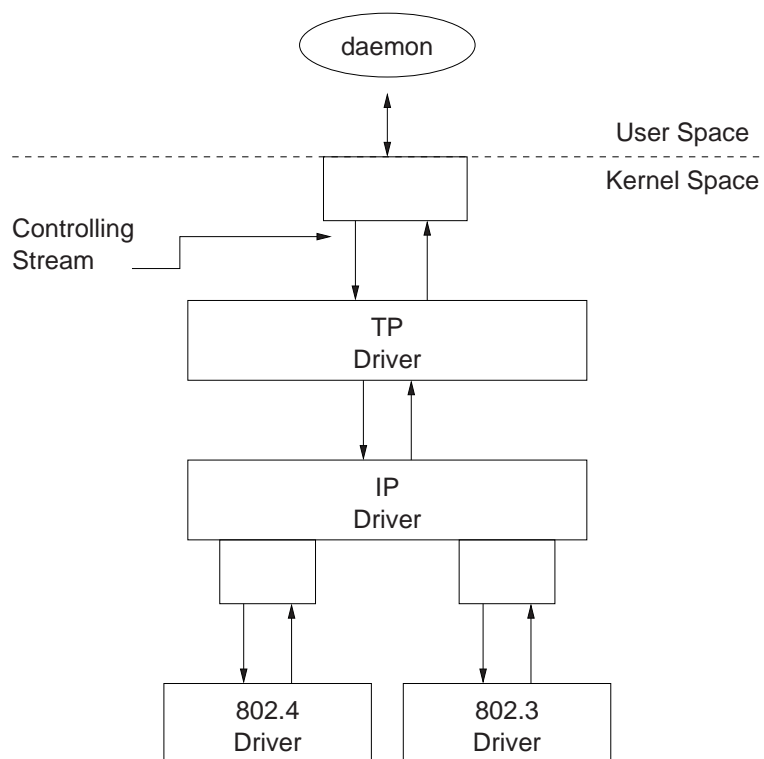


Figure 10.5: *TP Multiplexor*

Because the controlling *Stream* of the IP multiplexor has been linked below the TP multiplexor, the controlling *Stream* for the new multi-level multiplexor configuration is the *Stream* above the TP multiplexor.

At this point the file descriptors associated with the lower drivers can be closed without affecting the operation of the multiplexor. If these file descriptors are not closed, all

subsequent read, write, `ioctl`, poll, getmsg, and putmsg system calls issued to them will fail. That is because `I_LINK` associates the *Stream head* of each linked *Stream* with the multiplexor, so the user may not access that *Stream* directly for the duration of the link.

The following sequence of system calls completes the daemon example:

```

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

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

To summarize, [Figure 10.5](#) shows the multi-level protocol multiplexor. The transport driver supports several simultaneous *Streams*. These *Streams* are multiplexed over the single *Stream* connected to the IP multiplexor. The mechanism for establishing multiple *Streams* above the transport multiplexor is actually a by-product of the way in which *Streams* are created between a user process and a driver. By opening different minor devices of a *STREAMS* driver, separate *Streams* will be connected to that driver. Of course, the driver must be designed with the intelligence to route data from the single lower *Stream* to the appropriate upper *Stream*.

The daemon process maintains the multiplexed *Stream* configuration through an open *Stream* (the controlling *Stream*) to the transport driver. Meanwhile, other users can access the services of the transport protocol by opening new *Streams* to the transport driver; they are freed from the need for any unnecessary knowledge of the underlying protocol configurations and sub-networks that support the transport service.

Multi-level multiplexing configurations should be assembled from the bottom up. That is because the passing of `ioctls` through the multiplexor is determined by the nature of the multiplexing driver and cannot generally be relied on.

10.1.2 Dismantling a Multiplexor

Streams connected to a multiplexing driver from above with open, can be dismantled by closing each *Stream* with close. The mechanism for dismantling *Streams* that have been linked below a multiplexing driver is less obvious, and is described below.

The `I_UNLINK` `ioctl` call is used to disconnect each multiplexor link below a multiplexing driver individually. This command has the form:

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

where `fd` is a file descriptor associated with a *Stream* connected to the multiplexing driver from above, and `muxid` is the identifier that was returned by `I_UNLINK` when a driver was linked below the multiplexor. Each lower driver may be disconnected individually in this way, or a special `muxid` value of -1 may be used to disconnect all drivers from the multiplexor simultaneously.

In the multiplexing daemon program presented earlier, the multiplexor is never explicitly dismantled. That is because all links associated with a multiplexing driver are automatically dismantled when the controlling *Stream* associated with that multiplexor is closed. Because the controlling *Stream* is open to a driver, only the final call of close for that *Stream* will

close it. In this case, the daemon is the only process that has opened the controlling *Stream*, so the multiplexing configuration will be dismantled when the daemon exits.

For the automatic dismantling mechanism to work in the multi-level, multiplexed *Stream* configuration, the controlling *Stream* for each multiplexor at each level must be linked under the next higher level multiplexor. In the example, the controlling *Stream* for the IP driver was linked under the TP driver. This resulted in a single controlling *Stream* for the full, multi-level configuration. Because the multiplexing program relied on closing the controlling *Stream* to dismantle the multiplexed *Stream* configuration instead of using explicit `I_UNLINK` calls, the muxid values returned by `I_LINK` could be ignored.

An important side effect of automatic dismantling on the close is that it is not possible for a process to build a multiplexing configuration with `I_LINK` and then exit. That is because `exit(2)` will close all files associated with the process, including the controlling *Stream*. To keep the configuration intact, the process must exist for the life of that multiplexor. That is the motivation for implementing the example as a daemon process.

However, if the process uses persistent links via the `I_PLINK ioctl` call, the multiplexor configuration would remain intact after the process exits. Persistent links are described later in this chapter.

10.1.3 Routing Data Through a Multiplexor

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

One routing option available to the multiplexor is to use the muxid value to determine to which *Stream* data should be routed (remember that each multiplexor link is associated with a muxid). `I_LINK` passes the muxid value to the driver and returns this value to the user. The driver can therefore specify that the muxid value must accompany data routed through it. For example, if a multiplexor routed data from a single upper *Stream* to one of several lower *Streams* (as did the IP driver), the multiplexor could require the user to insert the muxid of the desired lower *Stream* into the first four bytes of each message passed to it. The driver could then match the muxid in each message with the muxid of each lower *Stream*, and route the data accordingly.

10.2 Connecting and Disconnecting Lower Stream

Multiple *Streams* are created above a driver/multiplexor by use of the open system call on either different minor devices, or on a cloneable device file. Note that any driver that handles more than one minor device is considered an upper multiplexor.

To connect *Streams* below a multiplexor requires additional software within the multiplexor. The main difference between *STREAMS* lower multiplexors and *STREAMS* device drivers are that multiplexors are pseudo-devices and that multiplexors have two additional qinit structures, pointed to by fields in the `streamtab` structure: the lower half read-side qinit and the lower half write-side qinit.

The multiplexor is conceptually divided into two parts: the lower half (bottom) and the upper half (top). The multiplexor queue structures that have been allocated when the multiplexor was opened, use the usual qinit entries from the multiplexor's `streamtab`. This is the same as any open of the *STREAMS* device. When a lower *Stream* is linked beneath the multiplexor, the qinit structures at the *Stream head* are substituted by the bottom half qinit structures of the multiplexors. Once the linkage is made, the multiplexor switches messages between upper and lower *Streams*. When messages reach the top of the lower *Stream*, they are handled by `put` and `service` routines specified in the bottom half of the multiplexor.

10.2.1 Connecting Lower Streams

A lower multiplexor is connected as follows: the initial open to a multiplexing driver creates a *Stream*, as in any other driver. `open` uses the first two `streamtab` structure entries to create the driver queues. At this point, the only distinguishing characteristic of this *Stream* are non-‘NULL’ entries in the `streamtab` `st_muxrinit` and `st_muxwinit` fields.

These fields are ignored by `open` (see the rightmost *Stream* in [Figure 10.6](#)). Any other *Stream* subsequently opened to this driver will have the same `streamtab` and thereby the same mux fields.

Next, another file is opened to create a (soon to be) lower *Stream*. The driver for the lower *Stream* is typically a device driver (see the leftmost *Stream* in [Figure 10.6](#)). This *Stream* has no distinguishing characteristics. It can include any driver compatible with the multiplexor. Any modules required on the lower *Stream* must be pushed onto it now.

Next, this lower *Stream* is connected below the multiplexing driver with an `I_LINK ioctl` call [see `streamio(7)`]. The *Stream head* points to the *Stream head* routines as its procedures (known via its queue). An `I_LINK` to the upper *Stream*, referencing the lower *Stream*, causes *STREAMS* to modify the contents of the *Stream head*'s queues in the lower *Stream*. The pointers to the *Stream head* routines, and other values, in the *Stream head*'s queues are replaced with those contained in the mux fields of the multiplexing driver's `streamtab`. Changing the *Stream head* routines on the lower *Stream* means that all subsequent messages sent upstream by the lower *Stream*'s driver will, ultimately, be passed to the `put` procedure designated in `st_muxrinit`, the multiplexing driver. The `I_LINK` also establishes this upper *Stream* as the control *Stream* for this lower *Stream*. *STREAMS* remembers the relationship between these two *Streams* until the upper *Stream* is closed, or the lower *Stream* is unlinked.

Finally, the *Stream head* sends an `M_IOCTL` message with `ioc_cmd` set to `I_LINK` to the multiplexing driver. The `M_DATA` part of the `M_IOCTL` contains a `linkblk` structure. The multiplexing driver stores information from the `linkblk` structure in private storage and returns an `M_IOCACK` message (acknowledgement). `l_index` is returned to the process requesting the `I_LINK`. This value can be used later by the process to disconnect this *Stream*. An `I_LINK` is required for each lower *Stream* connected to the driver. Additional upper *Streams* can be connected to the multiplexing driver by `open` calls. Any message type can be sent from a lower *Stream* to user processes along any of the upper *Streams*. The upper *Streams* provide the only interface between the user processes and the multiplexor.

Note that no direct data structure linkage is established for the linked *Streams*. The read queue's `q_next` will be ‘NULL’ and the write queue's `q_next` will point to the first entity

on the lower *Stream*. Messages flowing upstream from a lower driver (a device driver or another multiplexor) will enter the multiplexing driver `put` procedure with `Lqbot` as the queue value. The multiplexing driver has to route the messages to the appropriate upper (or lower) *Stream*. Similarly, a message coming downstream from user space on any upper *Stream* has to be processed and routed, if required, by the driver.

Also note that the lower *Stream* (see the headers and file descriptors in [Figure 10.7](#)) is no longer accessible from user space. This causes all system calls to the lower *Stream* to return `[EINVAL]`, with the exception of `close`. This is why all modules have to be in place before the lower *Stream* is linked to the multiplexing driver.

Finally, note that the absence of direct linkage between the upper and lower *Streams* means that *STREAMS* flow control has to be handled by special code in the multiplexing driver. The flow control mechanism cannot see across the driver.

In general, multiplexing drivers should be implemented so that new *Streams* can be dynamically connected to (and existing *Streams* disconnected from) the driver without interfering with its ongoing operation. The number of *Streams* that can be connected to a multiplexor is developer dependent.

10.2.2 Disconnection Lower Streams

Dismantling a lower multiplexor is accomplished by disconnecting (unlinking) the lower *Streams*. Unlinking can be initiated in three ways: an `I_UNLINK ioctl` referencing a specific *Stream*, an `I_UNLINK` indicating all lower *Streams*, or the last `close` of the control *Stream*. As in the link, an unlink sends a `linkblk` structure to the driver in an `M_IOCTL` message. The `I_UNLINK` call, which unlinks a single *Stream*, uses the `L_index` value returned in the `I_LINK` to specify the lower *Stream* to be unlinked. The latter two calls must designate a file corresponding to a control *Stream* which causes all the lower *Streams* that were previously linked by this control *Stream* to be unlinked. However, the driver sees a series of individual unlinks.

If no open references exist for a lower *Stream*, a subsequent unlink will automatically close the *Stream*. Otherwise, the lower *Stream* must be closed by `close` following the unlink. *STREAMS* will automatically dismantle all cascaded multiplexors (below other multiplexing *Streams*) if their controlling *Stream* is closed. An `I_UNLINK` will leave lower, cascaded multiplexing *Streams* intact unless the *Stream* file descriptor was previously closed.

10.3 Multiplexor Construction Example

This section describes an example of multiplexor construction and usage. [Figure 10.6](#) shows the *Streams* before their connection to create the multiplexing configuration of [Figure 10.7](#). Multiple upper and lower *Streams* interface to the multiplexor driver. The user processes of [Figure 10.7](#) are not shown in [Figure 10.6](#).

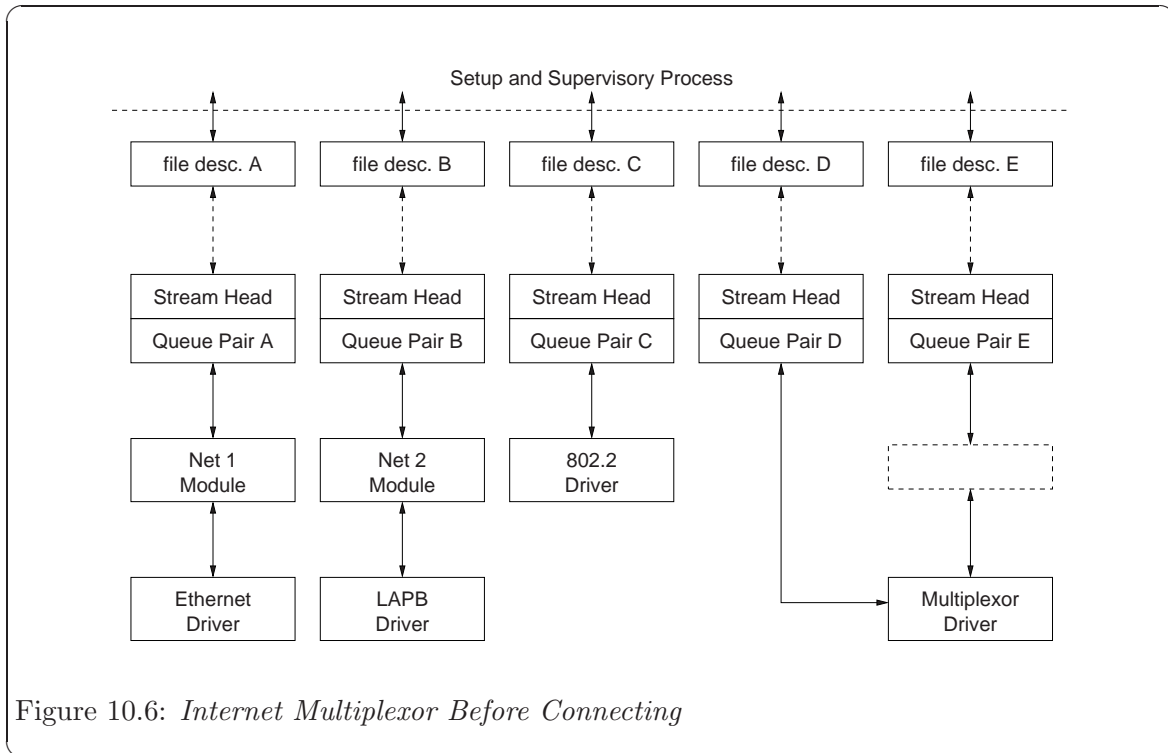
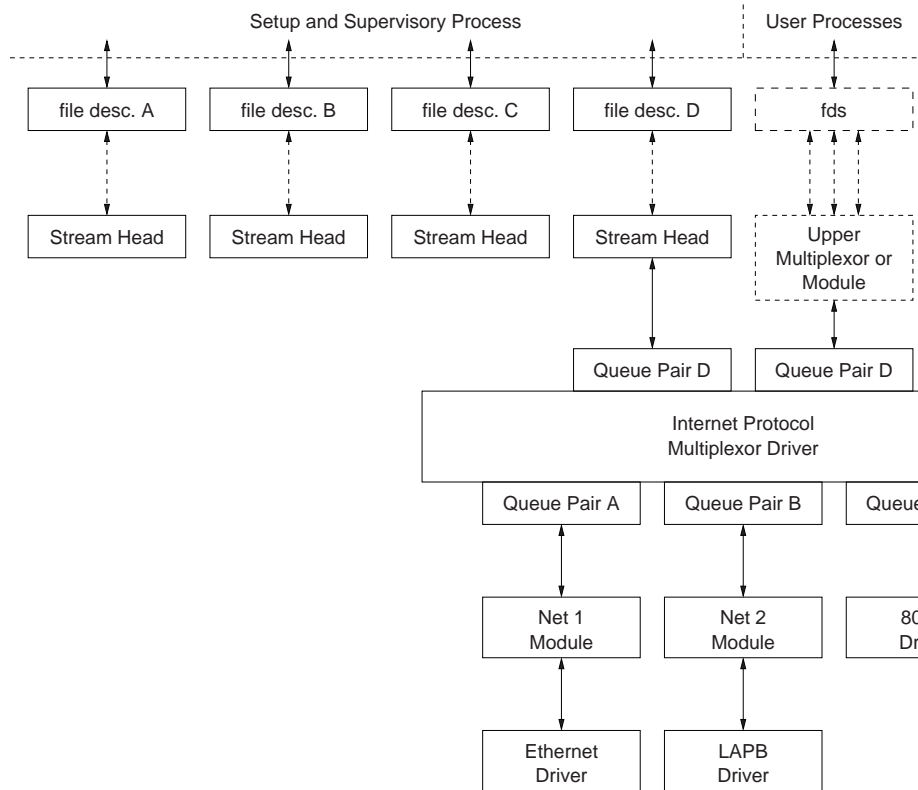


Figure 10.6: *Internet Multiplexor Before Connecting*

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

Figure 10.6 depicts only a portion of the full, larger *Stream*. In the dotted rectangle above the IP multiplexor, there generally would be an upper transport control protocol (*TCP*) multiplexor, additional modules and, possibly, additional multiplexors in the *Stream*. Multiplexors could also be cascaded below the IP driver if the device drivers were replaced by multiplexor drivers.

Figure 10.7: *Internet Multiplexor After Connecting*

Streams A, B, and C are opened by the process, and modules are pushed as needed. Two upper *Streams* are opened to the IP multiplexor. The rightmost *Stream* represents multiple *Streams*, each connected to a process using the network. The *Stream* second from the right provides a direct path to the multiplexor for supervisory functions. It is the control *Stream*, leading to a process which sets up and supervises this configuration. It is always directly connected to the IP driver. Although not shown, modules can be pushed on the control *Stream*.

After the *Streams* are opened, the supervisory process typically transfers routing information to the IP drivers (and any other multiplexors above the IP), and initializes the links. As each link becomes operational, its *Stream* is connected below the IP driver. If a more complex multiplexing configuration is required, the IP multiplexor *Stream* with all its connected links can be connected below another multiplexor driver.

Figure 10.7 shows that the file descriptors for the lower device driver *Streams* are left dangling. The primary purpose in creating these *Streams* was to provide parts for the multiplexor. Those not used for control and not required for error recovery (by reconnecting them through an `I_UNLINK ioctl`) have no further function. These lower *Streams* can be closed to free the file descriptor without any effect on the multiplexor.

10.4 Multiplexing Driver

This section contains an example of a multiplexing driver that implements an N-to-1 configuration. This configuration might be used for terminal windows, where each transmission to or from the terminal identifies the window. This resembles a typical device driver, with two differences: the device handling functions are performed by a separate driver, connected as a lower *Stream*, and the device information (i.e., relevant user process) is contained in the input data rather than in an interrupt call.

Each upper *Stream* is created by `open(2)`. A single lower *Stream* is opened and then it is linked by use of the multiplexing facility. This lower *Stream* might connect to the *tty* driver. The implementation of this example is a foundation for an M-to-N multiplexor.

As in the loop-around driver (in Drivers), flow control requires the use of standard and special code, since physical connectivity among the *Streams* is broken at the driver. Different approaches are used for flow control on the lower *Stream*, for messages coming upstream from the device driver, and on the upper *Streams*, for messages coming downstream from the user processes.

The multiplexor declarations are:

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/streams.h>
#include <sys/stropts.h>
#include <sys/errno.h>
#include <sys/cred.h>
#include <sys/ddi.h>

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

static struct module_info info = {
    0xaabb, "mux", 0, INFPSZ, 512, 128
};

static struct qinit urinit = { /* upper read */
    NULL, NULL, muxopen, muxclose, NULL, &info, NULL
};

static struct qinit uwinit = { /* upper write */
    muxuwput, muxuwsrv, NULL, NULL, NULL, &info, NULL
};

static struct qinit lrinit = { /* lower read */
    muxlrput, NULL, NULL, NULL, NULL, &info, NULL
};

static struct qinit lwinit = { /* lower write */
    NULL, muxlwsrv, NULL, NULL, NULL, &info, NULL
};

struct streamtab muxinfo = { &urinit, &uwinit, &lrinit, &lwinit
};
```

```

struct mux {
    queue_t *qbtr; /* back pointer to read queue */
};

extern struct mux mux_mux[];
extern int mux_cnt;

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

```

The four `streamtab` entries correspond to the upper read, upper write, lower read, and lower write qinit structures. The multiplexing qinit structures replace those in each (in this case there is only one) lower *Stream head* after the `I_LINK` has completed successfully. In a multiplexing configuration, the processing performed by the multiplexing driver can be partitioned between the upper and lower queues. There must be an upper *Stream* write `put` procedure and lower *Stream* read `put` procedure. If the queue procedures of the opposite upper/lower queue are not needed, the queue can be skipped over, and the message put to the following queue.

In the example, the upper read-side procedures are not used. The lower *Stream* read queue `put` procedure transfers the message directly to the read queue upstream from the multiplexor. There is no lower write `put` procedure because the upper write `put` procedure directly feeds the lower write queue downstream from the multiplexor.

The driver uses a private data structure, `mux`. `mux_mux[dev]` points back to the opened upper read queue. This is used to route messages coming upstream from the driver to the appropriate upper queue. It is also used to find a free major/minor device for a `CLONEOPEN` driver open case.

The upper queue open contains the canonical driver open code:

```

static int
muxopen(q, devp, flag, sflag, credp)
    queue_t *q;
    dev_t *devp;
    int flag;
    int sflag;
    cred_t *credp;
{
    struct mux *mux;
    dev_t device;

    if (sflag == CLONEOPEN) {
        for (device = 0; device < mux_cnt; device++) {
            if (mux_mux[device].qptra == 0)
                break;
        }
    } else
        device = getminor(*devp);

    if (device >= mux_cnt)
        return ENXIO;

    mux = &mux_mux[device];
    mux->qptra = q;
}

```

```

    q->q_ptr = (char *) mux;
    wr(q)->q_ptr = (char *) mux;
    return 0;
}

```

`muxopen` checks for a clone or ordinary open call. It initializes `q_ptr` to point at the `mux_mux[t]` structure.

The core multiplexor processing is the following: downstream data written to an upper *Stream* is queued on the corresponding upper write message queue if the lower *Stream* is flow controlled. This allows flow control to propagate towards the *Stream head* for each upper *Stream*. A lower write **service** procedure, rather than a write **put** procedure, is used so that flow control, coming up from the driver below, may be handled.

On the lower read-side, data coming up the lower *Stream* are passed to the lower read **put** procedure. The procedure routes the data to an upper *Stream* based on the first byte of the message. This byte holds the minor device number of an upper *Stream*. The **put** procedure handles flow control by testing the upper *Stream* at the first upper read queue beyond the driver. That is, the **put** procedure treats the *Stream* component above the driver as the next queue.

10.4.1 Upper Write Put Procedure

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

```

static int
muxuwput(q, mp)
    queue_t *q;
    mblk_t *MP;
{
    int s;
    struct mux *mux;

    mux = (struct mux *) q->q_ptr;
    switch (mp->b_datap->db_type) {
    case M_IOCTL:
    {
        struct iocblk *iocp;
        struct linkblk *linkp;

        /*
         * ioctl. Only channel 0 can do ioctls. Two
         * calls are recognized: Link, and UNLINK
         */

        if (mux != mux_mux)
            goto iocnak;

        iocp = (struct iocblk *) mp->b_rptr;
        switch (iocp->ioc_cmd) {
        case I_LINK:

            /*
             * Link. The data contains a linkblk structure

```

```

        * Remember the bottom queue in muxbot.
        */

        if (muxbot != NULL)
            goto iocnak;
        linkp = (struct linkblk *) mp->b_cont->b_rptr;
        muxbot = linkp->l_qbot;
        muxerr = 0;
        mp->b_datap->db_type = M_IOCACK;
        iocp->ioc_count = 0;
        qreply(q, mp);
        break;

    case I_UNLINK:

        /*
         * Unlink. The data contains a linkblk structure.
         * Should not fail an unlink. Null out muxbot.
         */

        linkp = (struct linkblk *) mp->b_cont->b_rptr;
        muxbot = NULL;
        mp->b_datap->db_type = M_IOCACK;
        iocp->ioc_count = 0;
        qreply(q, mp);
        break;

    default:
        iocnak:

        /* fail ioctl */

        mp->b_datap->db_type = M_IOCNAK;
        qreply(q, mp);
    }

    break;
}

```

First, there is a check to enforce that the *Stream* associated with minor device 0 will be the single, controlling *Stream*. The *ioctl*s are only accepted on this *Stream*. As described previously, a controlling *Stream* is the one that issues the *I_LINK*. Having a single control *Stream* is a recommended practice. *I_LINK* and *I_UNLINK* include a *linkblk* structure containing:

<i>l_qtop</i>	The upper write queue from which the <i>ioctl</i> is coming. It should always equal <i>q</i> .
<i>l_qbot</i>	The new lower write queue. It is the former <i>Stream head</i> write queue. It is of most interest since that is where the multiplexor gets and puts its data.
<i>l_index</i>	A unique (system wide) identifier for the link. It can be used for routing or during selective unlinks. Since the example only supports a single link, <i>l_index</i> is not used.

For `I_LINK`, `l_qbot` is saved in `muxbot` and a positive acknowledgment is generated. From this point on, until an `I_UNLINK` occurs, data from upper queues will be routed through `muxbot`. Note that when an `I_LINK` is received, the lower Stream has already been connected. This allows the driver to send messages downstream to perform any initialization functions. Returning an `M_IOCNAK` message (negative acknowledgment) in response to an `I_LINK` will cause the lower *Stream* to be disconnected.

The `I_UNLINK` handling code nulls out `muxbot` and generates a positive acknowledgment. A negative acknowledgment should not be returned to an `I_UNLINK`. The *Stream head* assures that the lower Stream is connected to a multiplexor before sending an `I_UNLINK M_IOCTL`.

`muxuwput` handles `M_FLUSH` messages as a normal driver would:

```

    case M_FLUSH:
        if (*mp->b_rptr & FLUSHW)
            flushq(q, FLUSHDATA);
        if (*mp->b_rptr & FLUSHR) {
            *mp->b_rptr &= ~FLUSHW;
            qreply(q, mp);
        } else
            freemsg(mp);
        break;
    case M_DATA:
        /*
         * Data. If we have no bottom queue --> fail
         * Otherwise, queue the data and invoke the lower
         * service procedure.
         */
        if (muxerr || muxbot == NULL)
            goto bad;
        if (canput(muxbot->q_next)) {
            mblk_t *bp;

            if ((bp = allocb(1, BPRI_MED)) == NULL) {
                putq(q, mp);
                bufcall(1, BPRI_MED, qenable, q);
                break;
            }
            *bp->b_wptr++ = (struct mux *) q->q_ptr - mux_mux;
            bp->b_cont = mp;
            putnext(muxbot, bp);
        } else
            putq(q, mp);
        break;
    default:
        bad:
        /*
         * Send an error message upstream.
         */
        mp->b_datap->db_type = M_ERROR;
        mp->b_rptr = mp->b_wptr = mp->b_datap->db_base;
        *mp->b_wptr++ = EINVAL;
        qreply(q, mp);
    }
}

```

M_DATA messages are not placed on the lower write message queue. They are queued on the upper write message queue. When flow control subsides on the lower *Stream*, the lower **service** procedure, `muxlwsrv`, is scheduled to start output. This is similar to starting output on a device driver.

10.4.2 Upper Write Service Procedure

The following example shows the code for the upper multiplexor write **service** procedure:

```
static int
muxuwsrv(q)
    queue_t *q;
{
    struct mux *mulp;
    mblk_t *mp;

    mulp = (struct mux *) q->q_ptr;

    if (!muxbot && q->q_first) {
        flushq(q, FLUSHALL);
        return;
    }
    if (muxerr) {
        flushq(q, FLUSHALL);
        return;
    }
    while (mp = getq(q)) {
        if (canput(muxbot->q_next))
            putnext(muxbot, mp);
        else {
            putnext(muxbot, mp);
            return;
        }
    }
}
```

As long as there is a *Stream* still linked under the multiplexor and there are no errors, the **service** procedure will take a message off the queue and send it downstream, if flow control allows.

10.4.3 Lower Write Service Procedure

`muxlwsrv`, the lower (linked) queue write **service** procedure is scheduled as a result of flow control subsiding downstream (it is back-enabled).

```
static int
muxlwsrv(q)
    queue_t *q;
{
    register int i;

    for (i = 0; i < mux_cnt; i++)
        if (mux_mux[i].qptr && mux_mux[i].qptr->q_first)
            genable(mux_mux[i].qptr);
}
```

`mxlwsrv` steps through all possible upper queues. If a queue is active and there are messages on the queue, then its the upper write **service** procedure is enabled via `qenable(9)`.

10.4.4 Lower Read Put Procedure

The lower (linked) queue read put procedure is:

```
static int
mxlwrput(q, mp)
    queue_t *q;
    mklk_t *mp;
{
    queue_t *uq;
    mklk_t *b_cont;
    int device;

    if (muxerr) {
        freemsg(mp);
        return;
    }
    switch (mp->b_datap->db_type) {
    case M_FLUSH:
        /*
         * Flush queues. NOTE: sense of tests is reversed
         * since we are acting like a "stream head"
         */
        if (*mp->b_rptr & FLUSHW) {
            *mp->b_rptr &= ~FLUSHR;
            qreply(q, mp);
        } else
            freemsg(mp);
        break;

    case M_ERROR:
    case M_HANGUP:
        muxerr = 1;
        freemsg(mp);
        break;

    case M_DATA:
        /*
         * Route message. First byte indicates
         * device to send to. No flow control.
         *
         * Extract and delete device number. If the leading block is
         * now empty and more blocks follow, strip the leading block.
         */

        device = *mp->b_rptr++;

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

        if (device < 0 || device >= mux_cut) {
            freemsg(mp);
            break;
        }
    }
```

```

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

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

```

`muxlrput` receives messages from the linked *Stream*. In this case, it is acting as a *Stream head*. It handles `M_FLUSH` messages. Note the code is reversed from that of a driver, handling `M_FLUSH` messages from upstream. There is no need to flush the read queue because no data are ever placed on it.

`muxlrput` also handles `M_ERROR` and `M_HANGUP` messages. If one is received, it locks-up the upper *Streams* by setting `muxerr`.

`M_DATA` messages are routed by looking at the first data byte of the message. This byte contains the minor device of the upper *Stream*. Several sanity checks are made: Is the device in range? Is the upper *Stream* open? Is the upper *Stream* not full?

This multiplexor does not support flow control on the read-side. It is merely a router. If everything checks out, the message is put to the proper upper queue. Otherwise, the message is discarded.

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

```

/*
 * Upper queue close
 */
static int
muxclose(q, flag, credp)
    queue_t *q;
    int flag;
    cred_t *credp;
{
    ((struct mux *) q->q_ptr)->qp_ptr = NULL;
    q->q->ptr = NULL;
    wr(q)->q_ptr = NULL;
}

```

10.5 Persistent Links

With `I_LINK` and `I_UNLINK` ioctls the file descriptor associated with the *Stream* above the multiplexor used to set up the lower multiplexor connections must remain open for the duration of the configuration. Closing the file descriptor associated with the controlling *Stream* will dismantle the whole multiplexing configuration. Some applications may not

want to keep a process running merely to hold the multiplexor configuration together. Therefore, "free-standing" links below a multiplexor are needed. A persistent link is such a link. It is similar to a *STREAMS* multiplexor link, except that a process is not needed to hold the links together. After the multiplexor has been set up, the process may close all file descriptors and exit, and the multiplexor will remain intact.

Two `ioctl`s, `I_PLINK` and `I_PUNLINK`, are used to create and remove persistent links that are associated with the *Stream* above the multiplexor. `close(2)` and `I_UNLINK` are not able to disconnect the persistent links.

The format of `I_PLINK` is:

```
ioctl(fd0, I_PLINK, fd1)
```

The first file descriptor, `fd0`, must reference the *Stream* connected to the multiplexing driver and the second file descriptor, `fd1`, must reference the *Stream* to be connected below the multiplexor. The persistent link can be created in the following way:

```
upper_stream_fd = open("/dev/mux", O_RDWR);
lower_stream_fd = open("/dev/driver", O_RDWR);
muxid = ioctl(upper_stream_fd, I_PLINK, lower_stream_fd);
/*
 * save muxid in s file
 */
exit(0);
```

Figure 10.8 shows how `open(2)` establishes a *Stream* between the device and the *Stream* head.

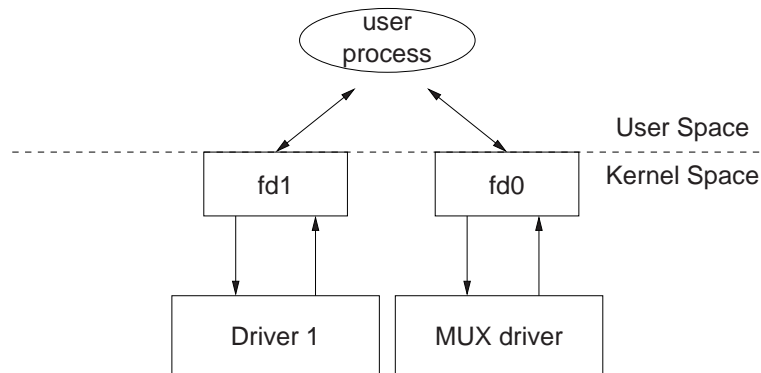


Figure 10.8: `open()` of *MUXdriver* and *Driver1*

The persistent link can still exist even if the file descriptor associated with the upper *Stream* to the multiplexing driver is closed. The `I_PLINK` `ioctl` returns an integer value, `muxid`, that can be used for dismantling the multiplexing configuration. If the process that created the persistent link still exists, it may pass the `muxid` value to some other process to dismantle the link, if the dismantling is desired, or it can leave the `muxid` value in a file so that other processes may find it later. Figure 10.9 shows a multiplexor after `I_PLINK`.

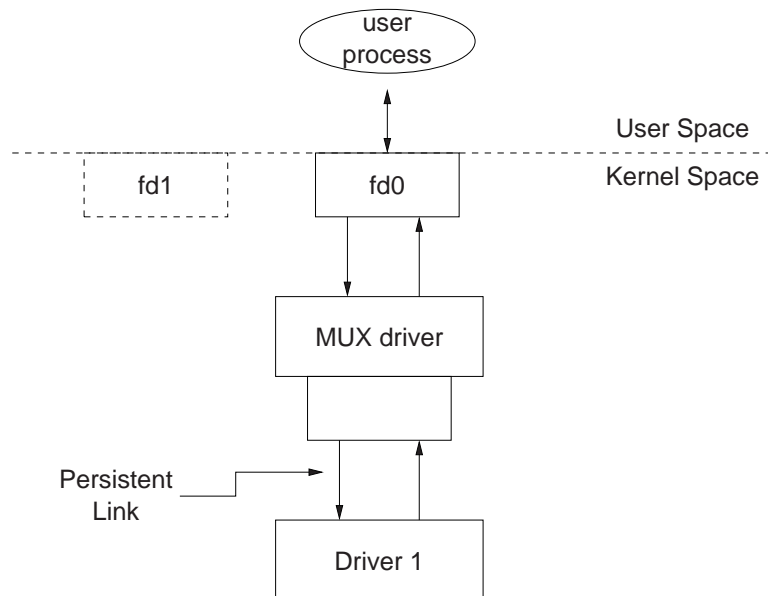


Figure 10.9: *Multiplexor After I_PLINK*

Several users can open the MUXdriver and send data to the Driver1 since the persistent link to the *Driver1* remains intact. This is shown in the following figure.

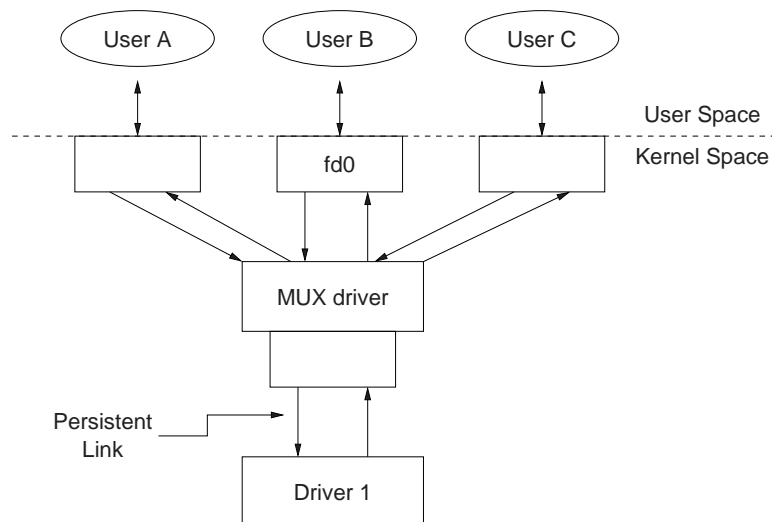


Figure 10.10: *Other Users Opening a MUXdriver*

The `ioctl I_PUNLINK` is used for dismantling the persistent link. Its format is:

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

where the *fd0* is the file descriptor associated with *Stream* connected to the multiplexing driver from above. The *muxid* is returned by the `ioctl I_PLINK` for the *Stream* that was connected below the multiplexor. The `I_PUNLINK` removes the persistent link between the multiplexor referenced by the *fd0* and the *Stream* to the driver designated by the *muxid*. Each of the bottom persistent links can be disconnected individually. An `I_PUNLINK ioctl` with the *muxid* value of `MUXID_ALL` will remove all persistent links below the multiplexing driver referenced by the *fd0*.

The following will dismantle the previously given configuration:

```
fd = open("/dev/mux", O_RDWR);
/*
 * retrieve muxid from the file
 */
ioctl(fd, I_PUNLINK, muxid);
exit(0);
```

The use of the `ioctls I_PLINK` and `I_PUNLINK` should not be intermixed with the `I_LINK` and `I_UNLINK`. Any attempt to unlink a regular link via the `I_PUNLINK` or to unlink a persistent link via the `I_UNLINK ioctl` will cause the `errno` value of `[EINVAL]` to be returned.

Since multi-level multiplexing configurations are allowed in *STREAMS*, it is possible to have a situation where persistent links exist below a multiplexor whose *Stream* is connected to the above multiplexor by regular links. Closing the file descriptor associated with the controlling *Stream* will remove the regular link but not the persistent links below it. On the other hand, regular links are allowed to exist below a multiplexor whose *Stream* is connected to the above multiplexor via persistent links. In this case, the regular links will be removed if the persistent link above is removed and no other references to the lower *Streams* exist.

The construction of cycles is not allowed when creating links. A cycle could be constructed by creating a persistent link of multiplexor 2 below multiplexor 1 and then closing the controlling file descriptor associated with the multiplexor 2 and reopening it again and then linking the multiplexor 1 below the multiplexor 2. This is not allowed. The operating system prevents a multiplexor configuration from containing a cycle to ensure that messages can not be routed infinitely, thus creating an infinite loop or overflowing the kernel stack.

10.6 Multiplexing Driver Design Guidelines

The following lists general multiplexor design guidelines:

- The upper half of the multiplexor acts like the end of the upper *Stream*.
- The lower half of the multiplexor acts like the head of the lower *Stream*.
- Service procedures are used for flow control.
- Message routing is based on multiplexor specific criteria.
- When one *Stream* is being fed by many *Streams*, flow control may have to take place. Then all feeding *Streams* on the other end of the multiplexor will have to be enabled when the flow control is relieved.
- When one *Stream* is feeding many *Streams*, flow control may also have to take place. Be careful not to starve other *Streams* when one becomes flow controlled.

11 STREAMS-based Pipes and FIFOs

11.1 Pipes and FIFOs

A pipe in the *UNIX* system is a mechanism that provides a communication path between multiple processes. Prior to *Release 4.0 UNIX System V* had "standard" pipes and named pipes (also called FIFOs). With standard pipes, one end was opened for reading and the other end for writing, thus data flow was uni-directional. FIFOs had only one end and typically one process opened the file for reading and another process opened the file for writing. Data written into the *FIFO* by the writer could then be read by the reader.

To provide greater support and development flexibility for applications using a network, pipes and FIFOs have become *STREAMS*-based in *UNIX System V Release 4.0*. The basic interface remains the same but the underlying implementation has changed. Pipes now provide a bi-directional mechanism for process communication. When a pipe is created via the `pipe(2)` system call, two *Streams* are opened and connected together, thus providing a full-duplex mechanism. Data flow is on First-In-First-Out basis. Previously pipes were associated with character devices and the creation of a pipe was limited to the capacity and configuration of the device. *STREAMS*-based pipes and FIFOs are not attached to *STREAMS*-based character devices. This eliminates configuration constraints and limits the number of opened pipes to the number of file descriptors for that process.

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

11.1.1 Creating and Opening Pipes and FIFOs

FIFOs are created via `mknod(2)` or `mkfifo(3C)`. FIFOs behave like regular file system nodes but are distinguished from other file system nodes by the `p` in the first column when the `ls -l` command is executed. Data written to the *FIFO* or read from the *FIFO* flow up and down the *Stream* in *STREAMS* buffers. Data written by one process can be read by another process.

FIFOs are opened in the same manner as other file system nodes via the `open(2)` system call. Any data written to the *FIFO* can be read from the same file descriptor in the First-In-First-Out manner. Modules can also be pushed on the *FIFO*. See `open(2)` for the restrictions that apply when opening a *FIFO*.

A *STREAMS*-based pipe is created via the `pipe(2)` system call that returns two file descriptors, `fd[0]` and `fd[1]`. Both file descriptors are opened for reading and writing. Data written to `fd[0]` becomes data read from `fd[1]` and vice versa.

Each end of the pipe has knowledge of the other end through internal data structures. Subsequent reads, writes, and closes are aware of if the other end of the pipe is open or closed. When one end of the pipe is closed, the internal data structures provide a way to access the *Stream* for the other end so that an `M_HANGUP` message can be sent to its *Stream head*.

After successful creation of a *STREAMS*-based pipe, 0 is returned. If `pipe(2)` is unable to create and open a *STREAMS*-based pipe, it will fail with `errno` set as follows:

- [ENOMEM] could not allocate two vnodes.
- [ENFILE] file table is overflowed.
- [EMFILE] can't allocate more file descriptors for the process.
- [ENOSR] could not allocate resources for both *Stream heads*.
- [EINTR] signal was caught while creating the *Stream heads*.

STREAMS modules can be added to a *STREAMS*-based pipe with the `ioctl(2)` `I_PUSH`. A module can be pushed onto one or both ends of the pipe (see [Figure 11.1](#)). However, a pipe maintains the concept of a midpoint so that if a module is pushed onto one end of the pipe, that module cannot be popped from the other end.

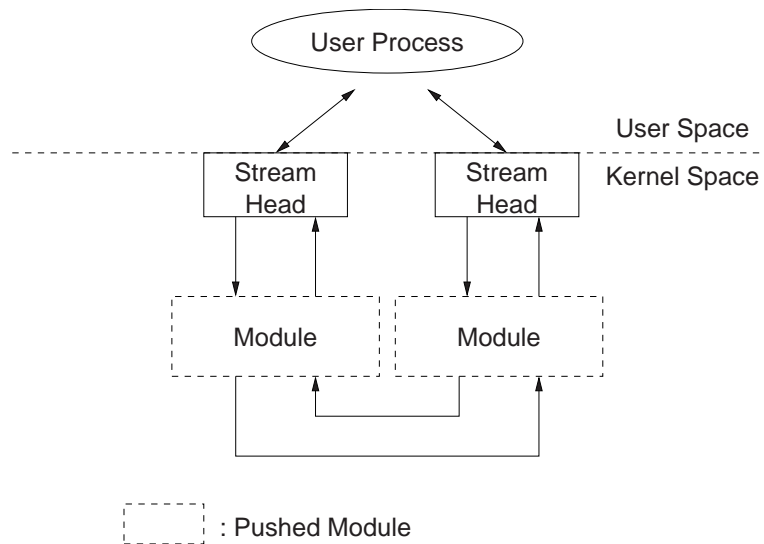


Figure 11.1: *Pushing Modules on a STREAMS-based Pipe*

11.1.2 Accessing Pipes and FIFOs

STREAMS-based pipes and FIFOs can be accessed through the operating system routines `read(2)`, `write(2)`, `ioctl(2)`, `close(2)`, `putmsg(2)`, `getmsg(2)`, and `poll(2)`. In case of FIFOs `open(2)` is also used.

11.1.2.1 Reading from a Pipe or FIFO

The `read(2)` [or `getmsg(2)`] system call is used to read from a pipe or *FIFO*. A user reads data from a *Stream* (not from a data buffer as was done prior to Release 4.0). Data can be read from either end of a pipe.

On success, the read returns the number of bytes read and placed in the buffer. When the end of the data is reached, the read returns 0.

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

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

11.1.2.2 Writing to a Pipe or FIFO

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

Zero Length Writes

If a user process issues `write(2)` with 0 as the number of bytes to send down a *STREAMS*-based pipe or *FIFO*, 0 is returned, and by default no message is sent down the *Stream*. However, if a user requires that a 0-length message be sent downstream, an `ioctl` call may be used to change this default behavior. The flag `SNDZERO` supports this. If `SNDZERO` is set in the *Stream head*, `write(2)` requests of 0 bytes will generate a 0-length message and send the message down the *Stream*. If `SNDZERO` is not set, no message is generated and 0 is returned to the user.

To toggle the `SNDZERO` bit, the `ioctl` `I_SWROPT` is used. If `arg` in the `ioctl` call is set to `SNDZERO` and the `SNDZERO` bit is off, the bit is turned on. If `arg` is set to 0 and the `SNDZERO` bit is on, the bit is turned off.

The `ioctl` `I_GWROPT` is used to return the current write settings.

Atomic Writes

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

`PIPE_BUF` is an implementation specific constant that specifies the maximum number of bytes that are atomic in a write to a pipe. When writing to a pipe, write requests of `PIPE_BUF` or less bytes will not be interleaved with data from other processes doing writes on the

same pipe. However, write requests greater than `PIPE_BUF` bytes may have data interleaved on arbitrary byte boundaries with writes by other processes whether or not the `O_NONBLOCK` or `O_NDELAY` flag is set.

If the module packet size is at least the size of `PIPE_BUF`, the *Stream head* packages the data in such a way that the first message is at least `PIPE_BUF` bytes. The remaining data may be packaged into smaller or larger blocks depending on buffer availability. If the first module on the Stream cannot support a packet of `PIPE_BUF`, atomic writes on the pipe cannot be guaranteed.

11.1.2.3 Closing a Pipe or FIFO

The `close(2)` system call closes a pipe or *FIFO* and dismantles its associated *Streams*. On the last close of one end of a pipe, an `M_HANGUP` message is sent upstream to the other end of the pipe. Subsequent `read(2)` or `getmsg(2)` calls on that *Stream head* will return the number of bytes read and zero when there are no more data. Subsequent `write(2)` or `putmsg(2)` requests will fail with `[ENXIO]`. If the pipe has been mounted via `fattach()` the pipe must be unmounted prior to calling `close`, otherwise the *Stream* will not be dismantled. If the other end of the pipe is mounted, the last close of the pipe will force it to be unmounted.

11.2 Flushing Pipes and FIFOs

When the flush request is initiated from a user `ioctl` or from a `flushq()` routine, the `FLUSHR` and/or `FLUSHW` bits of an `M_FLUSH` message will have to be switched. The point of switching the bits is the point where the `M_FLUSH` message is passed from a write queue to a read queue. This point is also known as the mid-point of the pipe.

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

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

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

The `pipemod` module can be pushed on any *Stream* that desires the bit switching. It must be pushed onto a pipe or *FIFO* if any form of flushing must take place.

11.3 Named Streams

Some applications may want to associate a *Stream* or *STREAMS*-based pipe with an existing node in the file system name space. For example, a server process may create a pipe, name one end of the pipe, and allow unrelated processes to communicate with it over that named end.

11.3.1 `fattach`

A *STREAMS* file descriptor can be named by attaching that file descriptor to a node in the file system name space. The routine `fattach()` [see also `fattach(3C)`] is used to name a *STREAMS* file descriptor. *Stream*, `fattach(3C)` Its format is:

```
int fattach (int fildes, char *path)
```

where `fildes` is an open file descriptor that refers to either a *STREAMS*-based pipe or a *STREAMS* device driver (or a pseudo device driver), and `path` is an existing node in the file system name space (for example, regular file, directory, character special file, etc).

The `path` cannot have a *Stream* already attached to it. It cannot be a mount point for a file system nor the root of a file system. A user must be an owner of the `path` with write permission or a user with the appropriate privileges in order to attach the file descriptor.

If the `path` is in use when the routine `fattach()` is executed, those processes accessing the `path` will not be interrupted and any data associated with the `path` before the call to the `fattach()` routine will continue to be accessible by those processes.

After a *Stream* is named, all subsequent operations [for example, `open(2)`] on the `path` will operate on the named *Stream*. Thus, it is possible that a user process has one file descriptor pointing to the data originally associated with the `path` and another file descriptor pointing to a named *Stream*.

Once the *Stream* has been named, the `stat(2)` system call on `path` will show information for the *Stream*. If the named *Stream* is a pipe, the `stat(2)` information will show that `path` is a pipe. If the *Stream* is a device driver or a pseudo device driver, `path` appears as a device. The initial modes, permissions, and ownership of the named *Stream* are taken from the attributes of the `path`. The user can issue the system calls `chmod(2)` and `chown(2)` to alter the attributes of the named *Stream* and not affect the original attributes of the `path` nor the original attributes of the *STREAMS* file.

The size represented in the `stat(2)` information will reflect the number of unread bytes of data currently at the *Stream head*. This size is not necessarily the number of bytes written to the *Stream*.

A *STREAMS*-based file descriptor can be attached to many different `paths` at the same time (i.e., a *Stream* can have many names attached to it). The modes, ownership, and permissions of these `paths` may vary, but operations on any of these `paths` will access the same *Stream*.

Named *Streams* can have modules pushed on them, be polled, be passed as file descriptors, and be used for any other *STREAMS* operation.

11.3.2 fdetach

A named *Stream* can be disassociated from a file name with the `fdetach()` routine [see also `fdetach(3C)`] that has the following format:

```
int fdetach (char *path)
```

where `path` is the name of the previously named *Stream*. Only the owner of `path` or the user with the appropriate privileges may disassociate the *Stream* from its name. The *Stream* may be disassociated from its name while processes are accessing it. If these processes have the named *Stream* open at the time of the `fdetach()` call, the processes will not get an error, and will continue to access the *Stream*. However, after the disassociation, subsequent operations on `path` access the underlying file rather than the named *Stream*.

If only one end of the pipe is named, the last close of the other end will cause the named end to be automatically detached. If the named *Stream* is a device and not a pipe, the last close will not cause the *Stream* to be detached.

If there is no named *Stream* or the user does not have access permissions on `path` or on the named *Stream*, `fdetach()` returns -1 with `errno` set to `[EINVAL]`. Otherwise, `fdetach()` returns 0 for success.

A *Stream* will remain attached with or without an active server process. If a server aborted, the only way a named *Stream* is cleaned up is if the server executed a clean up routine that explicitly detached and closed down the *Stream*.

If the named *Stream* is that of a pipe with only one end attached, clean up will occur automatically. The named end of the pipe is forced to be detached when the other end closes down. If there are no other references after the pipe is detached, the *Stream* is deallocated and cleaned up. Thus, a forced detach of a pipe end will occur when the server is aborted.

If the both ends of the pipe are named, the pipe remains attached even after all processes have exited. In order for the pipe to become detached, a server process would have to explicitly invoke a program that executed the `fdetach()` routine.

To eliminate the need for the server process to invoke the program, the `fdetach(1M)` command can be used. This command accepts a `path` name that is a path to a named *Stream*. When the command is invoked, the *Stream* is detached from the `path`. If the name was the only reference to the *Stream*, the *Stream* is also deallocated.

A user invoking the `fdetach(1M)` command must be an owner of the named *Stream* or a user with the appropriate permissions.

11.3.3 isastream

The function `isastream()` [see also `isastream(3C)`] may be used to determine if a file descriptor is associated with a *STREAMS* device. Its format is:

```
int isastream (int fildes)
```

where `fildes` refers to an open file. `isastream()` returns 1 if `fildes` represents a *STREAMS* file, and 0 if not. On failure, `isastream()` returns -1 with `errno` set to `[EBADF]`.

This function is useful for client processes communicating with a server process over a named *Stream* to check whether the file has been overlaid by a *Stream* before sending any data over the file.

11.3.4 File Descriptor Passing

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

11.3.5 Named Streams in A Remote Environment

If a user on the server machine creates a pipe and mounts it over a file that is part of an advertised resource, a user on the client machine (that has remotely named the resource) may access the remote named *Stream*. A user on the client machine is not allowed to pass file descriptors across the named *Stream* and will get an error when the `ioctl` request is attempted. If a user on the client machine creates a pipe and attempts to attach it to a file that is a remotely named resource, the system call will fail.

The following three examples are given as illustrations:

Suppose the server advertised a resource `/dev/foo`, created a *STREAMS*-based pipe, and attached one end of the pipe onto `/dev/foo/spipe`. All processes on the server machine will be able to access the pipe when they open `/dev/foo/spipe`. Now suppose that client ‘XYZ’ mounts the advertised resource `/dev/foo` onto its `/mnt` directory. All processes on client ‘XYZ’ will be able to access the *STREAMS*-based pipe when they open `/mnt/spipe`.

If the server advertised another resource `/dev/fog` and client ‘XYZ’ mounts that resource onto its `/install` directory and then attaches a *STREAMS*-based pipe onto `/install`, the mount would fail with `errno` set to `[EBUSY]`, because `/install` is already a mount point. If client ‘XYZ’ attached a pipe onto `/install/spipe`, the mount would also fail with `errno` set to `[EREMOTE]`, because the mount would require crossing an *RFS* (*Remote File System*) mount point.

Suppose the server advertised its `/usr/control` directory and client ‘XYZ’ mounts that resource onto its `/tmp` directory. The server now creates a *STREAMS*-based pipe and attaches one end over its `/usr` directory. When the server opens `/usr` it will access the pipe. On the other hand, when the client opens `/tmp` it will access what is in the server’s `/usr/control` directory.

11.4 Unique Connections

With named pipes, client processes may communicate with a server process via a module called `connld` that enables a client process to gain a unique, non-multiplexed connection to a server. The `connld` module can be pushed onto the named end of the pipe. If `connld` is pushed on the named end of the pipe and that end is opened by a client, a new pipe will be created. One file descriptor for the new pipe is passed back to a client (named *Stream*) as the file descriptor from the `open(2)` system call and the other file descriptor is passed to the server. The server and the client may now communicate through a new pipe.

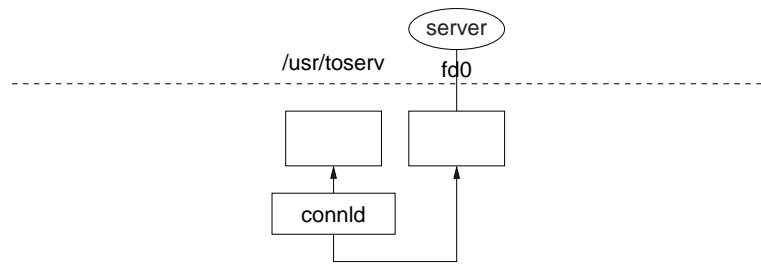
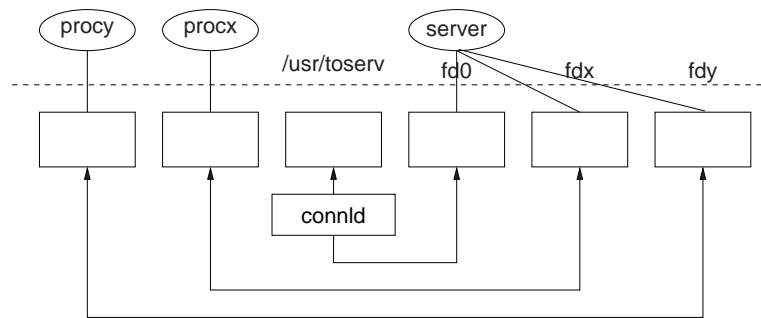
Figure 11.2: *Server Sets Up a Pipe*

Figure 11.2 illustrates a server process that has created a pipe and pushed the `connld` module on the other end. The server then invokes the `fattach()` routine to name the other end `/usr/toserv`.

Figure 11.3: *Processes X and Y Open /usr/toserv*

When process X (`procx`) opens `/usr/toserv`, it gains a unique connection to the server process that was at one end of the original *STREAMS*-based pipe. When process Y (`procy`) does the same, it also gains a unique connection to the server. As shown in Figure 11.3, the server process has access to three separate *STREAMS*-based pipes via three file descriptors.

`connld` is a *STREAMS*-based module that has an `open`, `close`, and `put` procedure. `connld` is opened when the module is pushed onto the pipe for the first time and whenever the named end of the pipe is opened. The `connld` module distinguishes between these two opens by use of the `q_ptr` field of its read queue. On the first open, this field is set to 1 and the routine returns without further processing. On subsequent opens, the field is checked for 1 or 0. If the 1 is present, the `connld` module creates a pipe and sends the file descriptor to a client and a server.

Making use of the `q_ptr` field eliminates the need to configure the `connld` module at boot time. It also eliminates the need to manage the number of times the module is either pushed and/or popped.

When the named *Stream* is opened, the open routine of `connld` is called. The `connld` open will fail if:

- The pipe ends can not be created.
- A file pointer and file descriptor can not be allocated.
- The *Stream head* can not stream the two pipe ends.
- `striotcl()` fails while sending the file descriptor to the server.

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

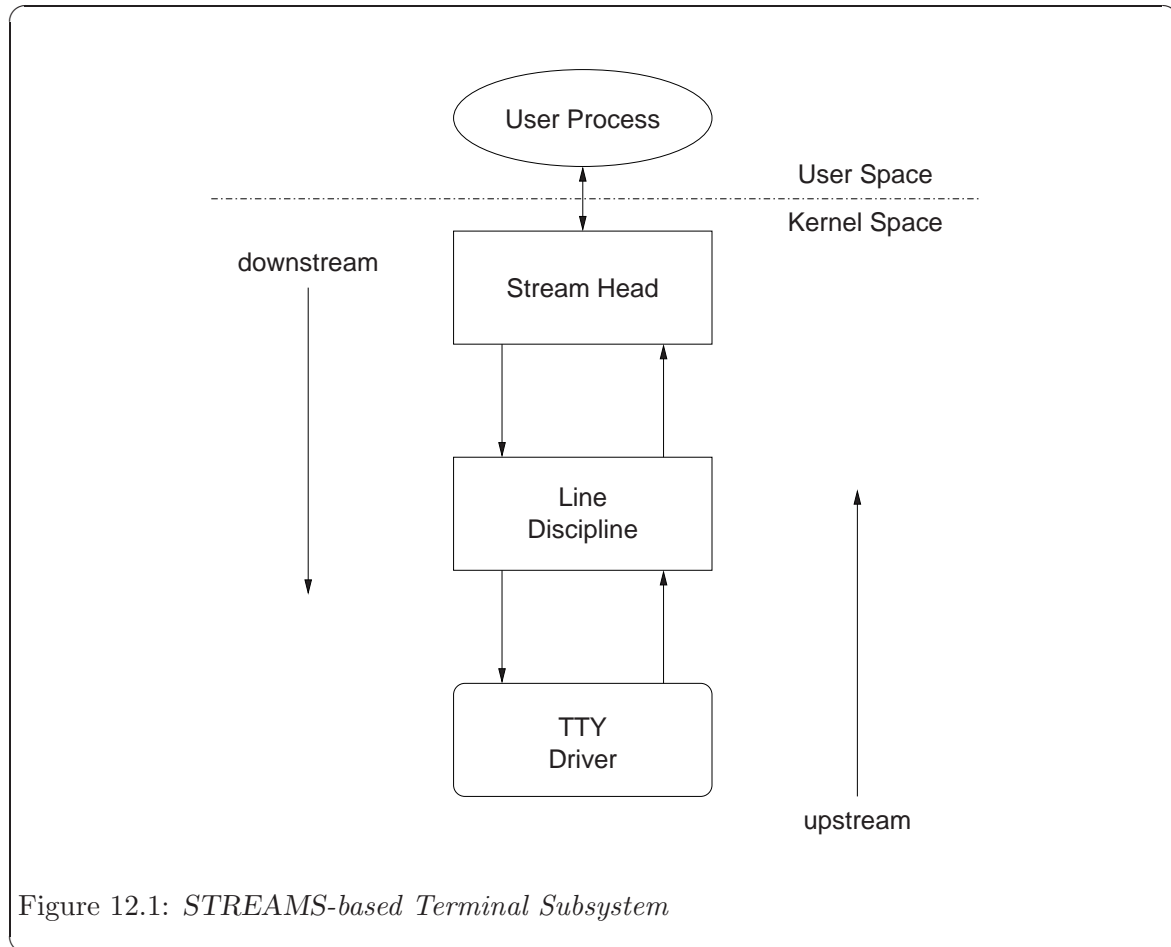
The `connld` module does not process messages. All messages are passed to the next object in the *Stream*. The read and write `put` routines call `putnext(9)` (see *STREAMS Utilities*) to send the message up or down the *Stream*.

12 STREAMS-based Terminal Subsystem

12.1 Terminal Subsystem

STREAMS provides a uniform interface for implementing character I/O devices and networking protocols in the kernel. *UNIX System V Release 4.0* implements the terminal subsystem in *STREAMS*. The *STREAMS*-based terminal subsystem (see [Figure 12.1](#)) provides many benefits:

- Reusable line discipline modules. The same module can be used in many *STREAMS* where the configuration of these *STREAMS* may be different.
- Line discipline substitution. Although *UNIX System V* provides a standard terminal line discipline module, another one conforming to the interface may be substituted. For example, a remote login feature may use the terminal subsystem line discipline module to provide a terminal interface to the user.
- Internationalization. The modularity and flexibility of the *STREAMS*-based terminal subsystem enables an easy implementation of a system that supports multiple byte characters for internationalization. This modularity also allows easy addition of new features to the terminal subsystem.
- Easy customizing. Users may customize their terminal subsystem environment by adding and removing modules of their choice.
- The pseudo-terminal subsystem. The pseudo-terminal subsystem can be easily supported (this is discussed in more detail later in this chapter).
- Merge with networking. By pushing a line discipline module on a network line, one can make the network look like a terminal line.

Figure 12.1: *STREAMS-based Terminal Subsystem*

The initial setup of the *STREAMS*-based terminal subsystem is handled with the `tty-mon(1M)` command within the framework of the Service Access Facility or the autopush feature. The autopush facility is discussed in [Appendix E \[STREAMS Configuration\]](#), page 323.

The *STREAMS*-based terminal subsystem supports `termio(7)`, the `termios(2)` specification of the POSIX standard, multiple byte characters for internationalization, the interface to asynchronous hardware flow control [see `termiox(7)`], and peripheral controllers for asynchronous terminals. *XENIX*® and *BSD* compatibility can also be provided by pushing the `ttcompat` module. *UNIX*® and *BSD* compatibility can also be provided by pushing the `ttcompat` module.

12.1.1 Line Discipline Module

A *STREAMS* line discipline module called `ldterm` [see `ldterm(7)`] is a key part of the *STREAMS*-based terminal subsystem. Throughout this chapter, the terms line discipline and `ldterm` are used interchangeably and refer to the *STREAMS* version of the standard line discipline and not the traditional character version. `ldterm` performs the standard terminal I/O processing which was traditionally done through the `linesw` mechanism.

The termio and termios specifications describe four flags which are used to control the terminal: `c_iflag` (defines input modes), `c_oflag` (defines output modes), `c_cflag` (defines hardware control modes), and `c_lflag` (defines terminal functions used by `ldterm`). In order to process these flags elsewhere (for example, in the firmware or in another process), a mechanism is in place to turn on and off the processing of these flags. When `ldterm` is pushed, it sends an `M_CTL` message downstream which asks the driver which flags the driver will process. The driver sends back that message in response if it needs to change `ldterm`'s default processing. By default, `ldterm` assumes that it must process all flags except `c_cflag`, unless it receives a message telling otherwise.

12.1.1.1 Default Settings

When `ldterm` is pushed on the *Stream*, the `open` routine initializes the settings of the termio flags. The default settings are:

```
c_iflag      '= BRKINT|ICRNL|IXON|ISTRIP|IXANY |ICRNL|IXON|ISTRIP|IXANY'
c_oflag      '= OPOST|ONLCR|TAB3 |ONLCR|TAB3'
c_cflag      '= 0'
c_lflag      '= ISIG|ICANON|ECHO|ECHOK |ICANON|ECHO|ECHOK'
```

In canonical mode (`ICANON` flag in `c_lflag` is turned on), read from the terminal file descriptor is in message non-discard (`RMSGN`) mode [see `streamio(7)`]. This implies that in canonical mode, read on the terminal file descriptor always returns at most one line regardless how many characters have been requested. In non-canonical mode, read is in byte-stream (`RNORM`) mode.

12.1.1.2 Data Structure

The `ldterm` module uses the following structure to maintain state information:

```
struct ldterm_mod {
    mblk_t *t_savbp;    /* saved mblk that holds ld structure */
    struct termios t_modes; /* effective modes set by the
                           provider */
    struct termios t_amodes; /* apparent modes for user programs */
    struct termios t_dmodes; /* modes that driver wishes to
                           process */
    unsigned long t_state; /* internal state of tty module */
    int t_line;    /* output line of tty */
    int t_col;    /* output column of tty */
    int t_rocount; /* number of characters echoed since last
                  output */
    int t_rocol;  /* column in which first such character
                  appeared */
    mblk_t *t_message; /* pointer to 1st mblk in message being built
                      */
    mblk_t *t_endmsg; /* pointer to last mblk in that message */
    int t_msglen;    /* number of characters in that message */
    mblk_t *t_echomp; /* echoed output being assembled */
    int t_rd_request; /* number of bytes requested by M_READ during
                    vmin/vtime read */
    int t_tid;    /* vtime timer id */
}
```

```

/*
 * The following are for EUC processing.
 */

unchar t_codeset; /* current code set indicator - read-side */
unchar t_eucleft; /* bytes left to get in current character */
unchar t_eucign; /* bytes left to ignore - output post proc */
unchar t_eucpad; /* padding for eucwioc */
eucioc_t eucwioc; /* eucioc structure (have to use bcopy) */
unchar *t_eucp; /* pointer to parallel array of column widths
                */
mbblk_t *t_eucp_mp; /* message block that holds parallel array */
unchar t_maxeuc; /* maximum length in memory bytes of an EUC */
int t_eucwarn; /* bad EUC counter */
};

```

12.1.1.3 Open and Close Routines

The open routine of the `ldterm` module allocates space for holding the `tty` structure (see ‘`tty.h`’) by allocating a buffer from the *STREAMS* buffer pool. The number of modules that can be pushed depends on the availability of buffers. The open also sends an `M_SETOPTS` message upstream to set the Stream head high and low water marks to 512 and 128 respectively.

The `ldterm` module establishes a controlling `tty` for the line when an `M_SETOPTS` message (`so_flags` is set to `SO_ISTTY`) is sent upstream. The *Stream head* allocates the controlling `tty` on the open, if one is not already allocated.

To maintain compatibility with existing applications that use the `O_NDELAY` flag, the open routine sets the `SO_NDELOK` flag on in the `so_flags` field of the `stroptions` structure in the `M_SETOPTS` message.

The open routine fails if there are no buffers available (cannot allocate the `tty` structure) or when an interrupt occurs while sleeping for a buffer to become available.

The close routine frees all the outstanding buffers allocated by this *Stream*. It also sends an `M_SETOPTS` message to the *Stream head* to undo the changes made by the open routine. The `ldterm` module also sends `M_START` and `M_STARTI` messages downstream to undo the effect of any previous `M_STOP` and `M_STOPI` messages.

12.1.1.4 Read-Side Processing

The `ldterm` module’s read-side processing has `put` and `service` procedures. High and low water marks for the read queue are 512 and 200 respectively.

`ldterm` can send the following messages upstream:

`ldterm(7)` messages

`M_DATA`, `M_BREAK`, `M_PCSIG`, `M_SIG`, `M_FLUSH`, `M_ERROR`, `M_IOCACK`, `M_IOCNAK`, `M_HANGUP`, `M_CTL`, `M_SETOPTS`, `M_COPYOUT`, and `M_COPYIN` (see [Section B.1 \[Message Types\]](#), page 281).

The `ldterm` module’s read-side processes `M_BREAK`, `M_DATA`, `M_CTL`, `M_FLUSH`, `M_HANGUP`, and `M_IOCACK` messages. All other messages are sent upstream unchanged.

The **put** procedure scans the message for flow control characters (**IXON**), signal generating characters, and after (possible) transformation of the message, queues the message for the **service** procedure. Echoing is handled completely by the **service** procedure.

In canonical mode if the **ICANON** flag is on in **c_lflag**, canonical processing is performed. If the **ICANON** flag is off, non-canonical processing is performed [see **termio(7)** for more details]. Handling of **VMIN/VTIME** in the *STREAMS* environment is somewhat complicated, because read needs to activate a timer in the **ldterm** module in some cases; hence, read notification becomes necessary. When a user issues an **ioctl** to **put** **ldterm** in non-canonical mode, the **ldterm** module sends an **M_SETOPTS** message to the *Stream head* to register read notification. Further reads on the terminal file descriptor will cause the *Stream head* to issue an **M_READ** message downstream and data will be sent upstream in response to the **M_READ** message. With read notification, buffering of raw data is performed by **ldterm**. It is possible to canonize the raw data, when the user has switched from raw to canonical mode. However, the reverse is not possible.

To summarize, in non-canonical mode, the **ldterm** module buffers all data until a request for the data arrives in the form of an **M_READ** message. The number of bytes sent upstream will be the argument of the **M_READ** message.

The **service** procedure of **ldterm** handles *STREAMS* related flow control. Since the read-side high and low water marks are 512 and 200 respectively, placing more than 512 characters on the **ldterm**'s read queue will cause the **QFULL** flag be turned on indicating that the module below should not send more data upstream.

Input flow control is regulated by the line discipline module by generating **M_STARTI** and **M_STOPI** high priority messages. When sent downstream, receiving drivers or modules take appropriate action to regulate the sending of data upstream. Output flow control is activated when **ldterm** receives flow control characters in its data stream. The **ldterm** module then sets an internal flag indicating that output processing is to be restarted/stopped and sends an **M_START/M_STOP** message downstream.

12.1.1.5 Write-Side Processing

Write-side processing of the **ldterm** module is performed by the write-side **put** procedures.

The **ldterm** module supports the following **ioctls**:

TCSETA, **TCSETAW**, **TCSETAF**, **TCSETS**, **TCSETSW**, **TCSETSF**, **TCGETA**, **TCGETS**, **TCXONC**, **TCFLSH**, **TCSBRK**, **TIOCSWINSZ**, **TIOCGWINSZ**, and **JWINSIZE**.

All **ioctls** not recognized by the **ldterm** module are passed downstream to the neighboring module or driver. *BSD* functionality is turned off by **IEXTEN** [see **termio(7)** for more details].

The following messages can be received on the write-side:

M_DATA, **M_DELAY**, **M_BREAK**, **M_FLUSH**, **M_STOP**, **M_START**, **M_STOPI**, **M_STARTI**, **M_READ**, **M_IOCDATA**, **M_CTL**, and **M_IOCTL**.

On the write-side, the **ldterm** module processes **M_FLUSH**, **M_DATA**, **M_IOCTL**, and **M_READ** messages, and all other message are passed downstream unchanged.

An **M_CTL** message is generated by **ldterm** as a query to the driver for an intelligent peripheral and to decide on the functional split for **termio** processing. If all or part of **termio**

processing is done by the intelligent peripheral, `ldterm` can turn off this processing to avoid computational overhead. This is done by sending an appropriate response to the `M_CTL` message, as follows: [see also `ldterm(7)`].

- If all of the termio processing is done by the peripheral hardware, the driver sends an `M_CTL` message back to `ldterm` with `ioc_cmd` of the structure `iocblk` set to `MC_NO_CANON`. If `ldterm` is to handle all termio processing, the driver sends an `M_CTL` message with `ioc_cmd` set to `MC_DO_CANON`. Default is `MC_DO_CANON`.
- If the peripheral hardware handles only part of the termio processing, it informs `ldterm` in the following way: The driver for the peripheral device allocates an `M_DATA` message large enough to hold a `termios` structure. The driver then turns on those `c_iflag`, `c_oflag`, and `c_lflag` fields of the `termios` structure that are processed on the peripheral device by ORing the flag values. The `M_DATA` message is then attached to the `b_cont` field of the `M_CTL` message it received. The message is sent back to `ldterm` with `ioc_cmd` in the data buffer of the `M_CTL` message set to `MC_PART_CANON`.

The line discipline module does not check if write-side flow control is in effect before forwarding data downstream. It expects the downstream module or driver to queue the messages on its queue until flow control is lifted.

12.1.1.6 EUC Handling in `ldterm`

The idea of letting post-processing (the `o_flags`) happen off the host processor is not recommended unless the board software is prepared to deal with international (*EUC*) character sets properly. The reason for this is that post-processing must take the *EUC* information into account. `ldterm` knows about the screen width of characters (that is, how many columns are taken by characters from each given code set on the current physical display) and it takes this width into account when calculating tab expansions. When using multi-byte characters or multi-column characters `ldterm` automatically handles tab expansion (when `TAB3` is set) and does not leave this handling to a lower module or driver.

As an example, consider the 3B2 PORTS board that has a processor and runs firmware on the board that can handle output post-processing. However, the firmware on the PORTS board has no knowledge of *EUC* unless one can change the firmware. Therefore, with some *EUC* code sets, particularly those where number of bytes in a character is not equivalent to the width of the character on the screen (for example, 3 byte codes that take only 2 screen columns), the PORTS board's firmware miscalculates the number of spaces required to expand the tab. Hence, if the board is allowed to handle tab expansion, it may get the expansion wrong in some cases.

By default multi-byte handling by `ldterm` is turned off. When `ldterm` receives an `EUC_WSET ioctl` call, it turns multi-byte processing on, if it is essential to properly handle the indicated code set. Thus, if one is using single byte 8-bit codes and has no special multi-column requirements, the special multi-column processing is not used at all. This means that multi-byte processing does not reduce the processing speed or efficiency of `ldterm` unless it is actually used.

The following describes how the *EUC* handling in `ldterm` works:

First, the multi-byte and multi-column character handling is only enabled when the `EUC_WSET ioctl` indicates that one of the following conditions is met:

- Code set consists of more than one byte (including the SS2 and/or SS3) of characters, or
- Code set requires more than one column to display on the current device, as indicated in the `EUC_WSET` structure.

Assuming that one or more of the above conditions, *EUC* handling is enabled. At this point, a parallel array (see `ldterm_mod` structure) used for other information, is allocated and a pointer to it is stored in `t_eucp_mp`. The parallel array which it holds is pointed to by `t_eucp`. The `t_codeset` field holds the flag that indicates which of the code sets is currently being processed on the read-side. When a byte with the high bit arrives, it is checked to see if it is SS2 or SS3. If so, it belongs to code set 2 or 3. Otherwise, it is a byte that comes from code set 1. Once the extended code set flag has been set, the input processor retrieves the subsequent bytes, as they arrive, to build one multi-byte character. The counter field `t_eucleft` tells the input processor how many bytes remain to be read for the current character. The parallel array `t_eucp` holds for each logical character in the canonical buffer its display width. During erase processing, positions in the parallel array are consulted to figure out how many backspaces need to be sent to erase each logical character. (In canonical mode, one backspace of input erases one logical character, no matter how many bytes or columns that character consumes.) This greatly simplifies erase processing for *EUC*.

The `t_maxeuc` field holds the maximum length, in memory bytes, of the *EUC* character mapping currently in use. The `eucwioc` field is a sub-structure that holds information about each extended code set.

The `t_eucign` field aids in output post-processing (tab expansion). When characters are output, `ldterm` keeps a column to indicate what the current cursor column is supposed to be. When it sends the first byte of an extended character, it adds the number of columns required for that character to the output column. It then subtracts one from the total width in memory bytes of that character and stores the result in `t_eucign`. This field tells `ldterm` how many subsequent bytes to ignore for the purposes of column calculation. (`ldterm` calculates the appropriate number of columns when it sees the first byte of the character.)

The field `t_eucwarn` is a counter for occurrences of bad extended characters. It is mostly useful for debugging.

There are two relevant files for handling multi-byte characters: `'euc.h'` and `'eucioctl.h'`. The `'eucioctl.h'` contains the structure that is passed with `EUC_WSET` and `EUC_WGET` calls. The normal way to use this structure is to get `CSWIDTH` (see note below) from the locale via a mechanism such as `getwidth` or `setlocale` and then copy the values into the structure in `'eucioctl.h'`, and send the structure via an `I_STR ioctl` call. The `EUC_WSET` call informs the `ldterm` module about the number of bytes in extended characters and how many columns the extended characters from each set consume on the screen. This allows `ldterm` to treat multi-byte characters as single entities for the purpose of erase processing and to correctly calculate tab expansions for multi-byte characters.

LC_CTYPE (instead of CSWIDTH) should be used in the environment in UXP/V systems. See `chrtbl(1M)` for more information.

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

```
#include <sys/euioctl.h>          /* need some other things too, like
                                   stropts.h */

struct eucioc eucw;              /* for EUC_WSET/EUC_WGET to line discipline */
eucwidth_t width;                /* return struct from_getwidth() */

/*
 * set_euc Send EUC code widths to line discipline.
 */
set_euc(e)
    struct eucioc *e;
{
    struct strioctl sb;

    sb.ic_cmd = EUC_WSET;
    sb.ic_timeout = 15;
    sb.ic_len = sizeof(struct eucioc);
    sb.ic_dp = (char *) e;

    if (ioctl(0, I_STR, &sb) < 0)
        fail();
}

/*
 * euclook Get current EUC code widths from line discipline.
 */
euclook(e)
    struct eucioc *e;
{
    struct strioctl sb;

    sb.ic_cmd = EUC_WGET;
    sb.ic_timeout = 15;
    sb.ic_len = sizeof(struct eucioc);
    sb.ic_dp = (char *) e;
    if (ioctl(0, I_STR, &sb) < 0)
        fail();

    printf("CSWIDTH=%d:%d, %d:%d, %d:%d0",
           e->eucw[1], e->scrw[1],
           e->eucw[2], e->scrw[2], e->eucw[3], e->scrw[3]);
}
```

12.1.2 Support of `termiox(7)`

The brief discussion of multiple byte character handling by the `ldterm` module was provided here for those interested in internationalization applications in UXP/V. More detailed descriptions may be obtained from product-related documents, for example, UXP/V Programmer's Guide: Internationalization.

UXP/V includes the extended general terminal interface [see `termiox(7)`] that supplements the `termio(7)` general terminal interface by adding for asynchronous hardware flow control, isochronous flow control and clock modes, and local implementations of additional asynchronous features. `termiox(7)` is handled by hardware drivers if the board (e.g., EPORTS) supports it.

Hardware flow control supplements the `termio(7)` `IXON`, `IXOFF`, and `IXANY` character flow control. The `termiox(7)` interface allows for both unidirectional and bidirectional hardware flow control. Isochronous communication is a variation of asynchronous communication where two communicating devices provide transmit and/or receive clock to each other. Incoming clock signals can be taken from the baud rate generator on the local isochronous port controller. Outgoing signals are sent on the receive and transmit baud rate generator on the local isochronous port controller.

Terminal parameters are specified in the `termiox` structure that is defined in the `'termiox.h'`.

12.1.3 Hardware Emulation Module

If a *Stream* supports a terminal interface, a driver or module that understands all `ioctl`s to support terminal semantics (specified by `termio` and `termios`) is needed. If there is no hardware driver that understands all `ioctl` commands downstream from the `ldterm` module, a hardware emulation module must be placed downstream from the line discipline module. The function of the hardware emulation module is to understand and acknowledge the `ioctl`s that may be sent to the process at the *Stream head* and to mediate the passage of control information downstream. The combination of the line discipline module and the hardware emulation module behaves as if there were an actual terminal on that *Stream*.

The hardware emulation module is necessary whenever there is no `tty` driver at the end of the *Stream*. For example, it is necessary in a *pseudo-tty* situation where there is process to process communication on one system (this is discussed later in this chapter) and in a network situation where a `termio` interface is expected (e.g., remote login) but there is no `tty` driver on the *Stream*.

Most of the actions taken by the hardware emulation module are the same regardless of the underlying architecture. However, there are some actions that are different depending on whether the communication is local or remote and whether the underlying transport protocol is used to support the remote connection. For example, *NTTY* is a hardware emulation module supported by AT&T in its Starla networking environment. This hardware emulation module behaves in a way understood by the *URP* protocol driver that exists below *NTTY*. On receipt of a `TCSBRK ioctl`, *NTTY* sends an `M_BREAK` message downstream. When the baud rate is 0, the hardware emulation module sends a *TPI* message requesting a disconnect. These actions are valid for a network situation but may not make sense in other environments when there is no module/driver below to understand the *TPI* messages or handle `M_BREAK` messages.

Each hardware emulation module has an open, close, read queue `put` procedure, and write queue `put` procedure.

The hardware emulation module does the following:

- Processes, if appropriate, and acknowledges receipt of the following `ioctl`s on its write queue by sending an `M_IOCACK` message back upstream: `TCSETA`, `TCSETAW`, `TCSETAF`, `TCSETS`, `TCSETSW`, `TCSETSF`, `TCGETA`, `TCGETS`, and `TCSBRK`.
- Acknowledges the Extended *UNIX* Code (*EUC*) `ioctl`s.
- If the environment supports windowing, it acknowledges the windowing `ioctl`s `TIOCSWINSZ`, `TIOCGWINSZ`, and `JWINSIZE`. If the environment does not support windowing, an `M_IOCNAK` message is sent upstream.
- If any other `ioctl`s are received on its write queue, it sends an `M_IOCNAK` message upstream.
- When the hardware emulation module receives an `M_IOCTL` message of type `TCSBRK` on its write queue, it sends an `M_IOCACK` message upstream and the appropriate message downstream. For example, an `M_BREAK` message could be sent downstream.
- When the hardware emulation module receives an `M_IOCTL` message on its write queue to set the baud rate to 0 (`TCSETAW` with `CBAUD` set to `B0`), it sends an `M_IOCACK` message upstream and an appropriate message downstream; for networking situations this will probably be an `M_PROTO` message which is a *TPI* `T_DISCON_REQ` message requesting the transport provider to disconnect.
- All other messages (`M_DATA`, etc.) not mentioned here are passed to the next module or driver in the *Stream*.

The hardware emulation module processes messages in a way consistent with the driver that exists below.

12.2 Pseudo-Terminal Subsystem

The *STREAMS*-based pseudo-terminal subsystem provides the user with an interface that is identical to the *STREAMS*-based terminal subsystem described earlier in this chapter. The pseudo-terminal subsystem (*pseudo-tty*) supports a pair of *STREAMS*-based devices called the master device and slave device. The slave device provides processes with an interface that is identical to the terminal interface. However, where all devices, which provide the terminal interface, have some kind of hardware device behind them, the slave device has another process manipulating it through the master half of the pseudo terminal. Anything written on the master device is given to the slave as an input and anything written on the slave device is presented as an input on the master side.

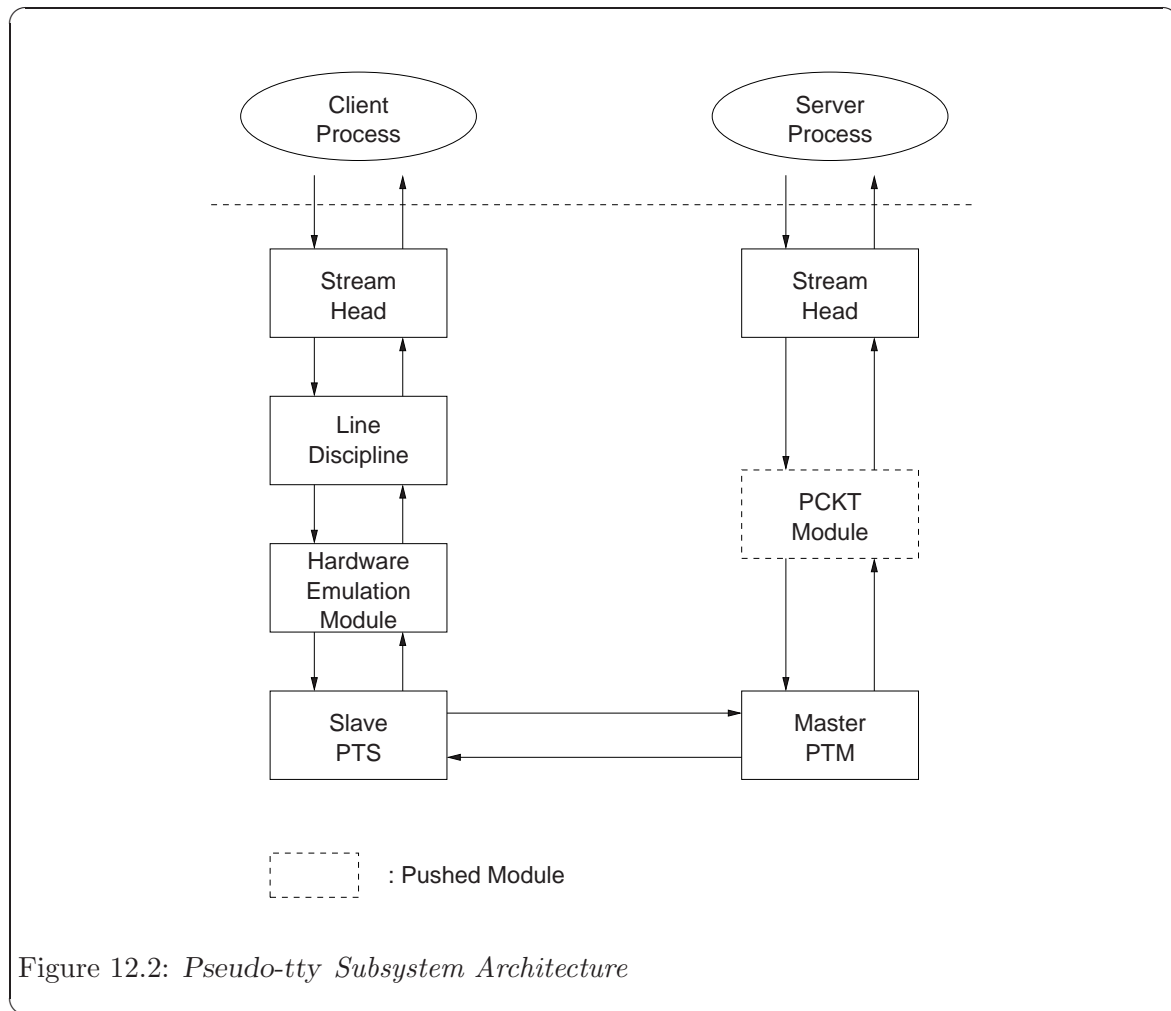
Figure 12.2 illustrates the architecture of the *STREAMS*-based pseudo-terminal subsystem. The master driver called `ptm` is accessed through the clone driver [see `clone(7)`] and is the controlling part of the system. The slave driver called `pts` works with the line discipline module and the hardware emulation module to provide a terminal interface to the user process. An optional packetizing module called `pkct` is also provided. It can be pushed on the master side to support packet mode (this is discussed later).

The number of *pseudo-tty* devices that can be installed on a system is dependent on available memory.

12.2.1 Line Discipline Module

In the *pseudo-tty* subsystem, the line discipline module is pushed on the slave side to present the user with the terminal interface.

`ldterm` may turn off the processing of the `c_iflag`, `c_oflag`, and `c_lflag` fields to allow processing to take place elsewhere. The `ldterm` module may also turn off all canonical processing when it receives an `M_CTL` message with the `MC_NO_CANON` command in order to support remote mode (this is discussed later). Although `ldterm` passes through messages without processing them, the appropriate flags are set when a "get" `ioctl`, such as `TCGETA` or `TCGETS`, is issued to indicate that canonical processing is being performed.



12.2.2 *Pseudo-tty* Emulation Module PTM

Since the *pseudo-tty* subsystem has no hardware driver downstream from the `ldterm` module to process the terminal `ioctl` calls, another module that understands the `ioctl` commands is placed downstream from the `ldterm`. This module, known as `ptem`, processes all of the terminal `ioctl` commands and mediates the passage of control information downstream.

ldterm and ptem together behave like a real terminal. Since there is no real terminal or modem in the *pseudo-tty* subsystem, some of the `ioctl` commands are ignored and cause only an acknowledgement of the command. The ptem module keeps track of the terminal parameters set by the various "set" commands such as `TCSETA` or `TCSETAW` but does not usually perform any action. For example, if one of the "set" `ioctls` is called, none of the bits in the `c_cflag` field of `termio` has any effect on the pseudo terminal except if the baud rate is set to 0. When setting the baud rate to 0, it has the effect of hanging up the pseudo-terminal.

The pseudo-terminal has no concept of parity so none of the flags in the `c_iflag` that control the processing of parity errors have any effect. The delays specified in the `c_oflag` field are not also supported.

The ptem module does the following:

- Processes, if appropriate, and acknowledges receipt of the following `ioctls` on its write queue by sending an `M_IOCACK` message back upstream: `TCSETA`, `TCSETAW`, `TCSETAF`, `TCSETS`, `TCSETSW`, `TCSETSF`, `TCGETA`, `TCGETS`, and `TCSBRK`.
- Keeps track of the window size; information needed for the `TIOCSWINSZ`, `TIOCGWINSZ`, and `JWINSIZE` `ioctl` commands.
- When it receives any other `ioctl` on its write queue, it sends an `M_IOCNAK` message upstream.
- It passes downstream the following `ioctls` after processing them: `TCSETA`, `TCSETAW`, `TCSETAF`, `TCSETS`, `TCSETSW`, `TCSETSF`, `TCSBRK`, and `TIOCSWINSZ`.
- ptem frees any `M_IOCNAK` messages it receives on its read queue in case the `pkct` module (`pkct` is described later) is not on the pseudo terminal subsystem and the above `ioctls` get to the master's *Stream head* which would then send an `M_IOCNAK` message.
- In its open routine, the ptem module sends an `M_SETOPTS` message upstream requesting allocation of a controlling *tty*.
- When the ptem module receives an `M_IOCTL` message of type `TCSBRK` on its read queue, it sends an `M_IOCACK` message downstream and an `M_BREAK` message upstream.
- When it receives an `ioctl` message on its write queue to set the baud rate to 0 (`TCSETAW` with `CBAUD` set to `B0`), it sends an `M_IOCACK` message upstream and a 0-length message downstream.
- When it receives an `M_IOCTL` of type `TIOCSIGNAL` on its read queue, it sends an `M_IOCACK` downstream and an `M_PCSIG` upstream where the signal number is the same as in the `M_IOCTL` message.
- When the ptem module receives an `M_IOCTL` of type `TIOCREMOTE` on its read queue, it sends an `M_IOCACK` message downstream and the appropriate `M_CTL` message upstream to enable/disable canonical processing.
- When it receives an `M_DELAY` message on its read or write queue, it discards the message and does not act on it.
- When it receives an `M_IOCTL` message with type `JWINSIZE` on its write queue and if the values in the `jwinsize` structure of ptem are not zero, it sends an `M_IOCACK` message

upstream with the `jwinsize` structure. If the values are zero, it sends an `M_IOCNAK` message upstream.

- When it receives an `M_IOCTL` message of type `TIOCGWINSZ` on its write queue and if the values in the `winsize` structure are not zero, it sends an `M_IOCACK` message upstream with the `winsize` structure. If the values are zero, it sends an `M_IOCNAK` message upstream. It also saves the information passed to it in the `winsize` structure and sends a *STREAMS* signal message for signal `{SIGWINCH}` upstream to the slave process if the size changed.
- When the `ptem` module receives an `M_IOCTL` message with type `TIOCGWINSZ` on its read queue and if the values in the `winsize` structure are not zero, it sends an `M_IOCACK` message downstream with the `winsize` structure. If the values are zero, it sends an `M_IOCNAK` message downstream. It also saves the information passed to it in the `winsize` structure and sends a *STREAMS* signal message for signal `{SIGWINCH}` upstream to the slave process if the size changed.
- All other messages not mentioned above are passed to the next module or driver.

12.2.2.1 Data Structure

Each instantiation of the `ptem` module is associated with a local area. These data are held in a structure called `ptem` that has the following format:

```
struct ptem {
    long cflags;           /* copy of c_flags */
    mblk_t *dack_ptr;      /* pointer to preallocated message block used
                           to send disconnect */
    queue_t *q_ptr;        /* pointer to ptem's read queue */
    struct winsize wsz;     /* structure to hold windowing information */
    unsigned short state;  /* state of ptem entry */
};
```

When the `ptem` module is pushed onto the slave side *Stream*, a search of the `ptem` structure is made for a free entry (state is not set to `INUSE`). The `c_cflags` of the `termio` structure and the windowing variables are stored in `cflags` and `wsz` respectively. The `dack_ptr` is a pointer to a message block used to send a 0-length message whenever a hang-up occurs on the slave side.

12.2.2.2 Open and Close Routines

In the open routine of `ptem` a *STREAMS* message block is allocated for a 0-length message for delivering a hang-up message; this allocation of a buffer is done before it is needed to ensure that a buffer is available. An `M_SETOPTS` message is sent upstream to set the read-side *Stream head* queues, to assign high and low water marks (512 and 256 respectively), and to establish a controlling terminal.

The default values `B300`, `CS8`, `CREAD`, and `HUPCL` are assigned to `cflags`, and `INUSE` to the state field.

The open routine fails if:

- No free entries are found when the `ptem` structure is searched.
- `sflag` is not set to `MODOPEN`.

- A 0-length message can not be allocated (no buffer is available).
- A `stroptions` structure cannot be allocated.

The close routine is called on the last close of the slave side *Stream*. Pointers to read and write queue are cleared and the buffer for the 0-length message is freed.

12.2.3 Remote Mode

A feature known as remote mode is available with the *pseudo-tty* subsystem. This feature is used for applications that perform the canonical function normally done by the `ldterm` module and *tty* driver. The remote mode allows applications on the master side to turn off the canonical processing. An `]` is issued on the master side to enter the remote mode. When this occurs, an `M_CTL` message with the command `MC_NO_CANON` is sent to the `ldterm` module indicating that data should be passed when received on the read-side and no canonical processing is to take place. The remote mode may be disabled by

```
ioctl(fd, TIOCREMOTE, 0).
```

12.2.4 Packet Mode

The *STREAMS*-based pseudo-terminal subsystem also supports a feature called packet mode. This is used to inform the process on the master side when state changes have occurred in the *pseudo-tty*. Packet mode is enabled by pushing the `pckt` module on the master side. Data written on the master side is processed normally. When data are written on the slave side or when other messages are encountered by the `pckt` module, a header is added to the message so it can be subsequently retrieved by the master side with a `getmsg` operation.

The `pckt` module does the following:

- When a message is passed to this module on its write queue, the module does noprocessing and passes the message to the next module or driver.
- The `pckt` module creates an `M_PROTO` message when one of the following messages is passed to it: `M_DATA`, `M_IOCTL`, `M_PROTO/M_PCPROTO`, `M_FLUSH`, `M_START/M_STOP`, `M_STARTI/M_STOPI`, and `M_READ`.

All other messages are passed through. The `M_PROTO` message is passed upstream and retrieved when the user issues `getmsg(2)`.

- If the message is an `M_FLUSH` message, `pckt` does the following: If the flag is `FLUSHW`, it is changed to `FLUSHR` (because `FLUSHR` was the original flag before the `pts` driver changed it), packetized into an `M_PROTO` message, and passed upstream. To prevent the *Stream head*'s read queue from being flushed, the original `M_FLUSH` message must not be passed upstream.

If the flag is `FLUSHR`, it is changed to `FLUSHW`, packetized into an `M_PROTO` message, and passed upstream. In order to flush of the write queues properly, an `M_FLUSH` message with the `FLUSHW` flag set is also sent upstream.

If the flag is `FLUSHRW`, the message with both flags set is packetized and passed upstream. An `M_FLUSH` message with the `FLUSHW` flag set is also sent upstream.

12.2.5 Pseudo-tty Drivers ptm and pts

In order to use the *pseudo-tty* subsystem, a node for the master side driver `/dev/ptmx` and `N` number of slave drivers (`N` is determined at installation time) must be installed. The names of the slave devices are `/dev/pts/M` where `M` has the values 0 through `N-1`. A user accesses a *pseudo-tty* device through the master device (called `ptm`) that in turn is accessed through the clone driver [see `clone(7)`]. The master device is set up as a clone device where its major device number is the major for the clone device and its minor device number is the major for the `ptm` driver.

The master pseudo driver is opened via the `open(2)` system call with `/dev/ptmx` as the device to be opened. The clone open finds the next available minor device for that major device; a master device is available only if it and its corresponding slave device are not already open. There are no nodes in the file system for master devices.

When the master device is opened, the corresponding slave device is automatically locked out. No user may open that slave device until it is unlocked. A user may invoke a function `grantpt` that will change the owner of the slave device to that of the user who is running this process, change the group id to `tty`, and change the mode of the device to `0620`. Once the permissions have been changed, the device may be unlocked by the user. Only the owner or super-user can access the slave device. The user must then invoke the `unlockpt` function to unlock the slave device. Before opening the slave device, the user must call the `ptsname` function to obtain the name of the slave device. The functions `grantpt`, `unlockpt`, and `ptsname` are called with the file descriptor of the master device. The user may then invoke the `open` system call with the name that was returned by the `ptsname` function to open the slave device.

The following example shows how a user may invoke the *pseudo-tty* subsystem:

```
int fdm, fds;
char *slavename;
extern char *ptsname();

fdm = open("/dev/ptmx", O_RDWR); /* open master */
grantpt(fdm);                  /* change permission of slave */
unlockpt(fdm);                 /* unlock slave */
slavename = ptsname(fdm);      /* get name of slave */
fds = open(slavename, O_RDWR); /* open slave */
ioctl(fds, I_PUSH, "ptem");    /* push ptem */
ioctl(fds, I_PUSH, "ldterm");  /* push ldterm */
```

Unrelated processes may open the pseudo device. The initial user may pass the master file descriptor using a *STREAMS*-based pipe or a slave name to another process to enable it to open the slave. After the slave device is open, the owner is free to change the permissions. Certain programs such as `write` and `wall` are set group-id (`setgid`) to `tty` and are also able to access the slave device.

After both the master and slave have been opened, the user has two file descriptors which provide full duplex communication using two *Streams*. The two *Streams* are automatically connected. The user may then push modules onto either side of the *Stream*. The user also needs to push the `ptem` and `ldterm` modules onto the slave side of the pseudo-terminal subsystem to get terminal semantics.

The master and slave drivers pass all *STREAMS* messages to their adjacent queues. Only the `M_FLUSH` needs some processing. Because the read queue of one side is connected to the write queue of the other, the `FLUSHR` flag is changed to `FLUSHW` flag and vice versa.

When the master device is closed, an `M_HANGUP` message is sent to the slave device which will render the device unusable. The process on the slave side gets the `errno` `[ENXIO]` when attempting to write on that *Stream* but it will be able to read any data remaining on the *Stream head* read queue. When all the data have been read, `read` returns 0 indicating that the *Stream* can no longer be used.

On the last close of the slave device, a 0-length message is sent to the master device. When the application on the master side issues a `read` or `getmsg` and 0 is returned, the user of the master device decides whether to issue a close that dismantles the pseudo-terminal subsystem. If the master device is not closed, the *pseudo-tty* subsystem will be available to another user to open the slave device.

Since 0-length messages are used to indicate that the process on the slave side has closed and should be interpreted that way by the process on the master side, applications on the slave side should not write 0-length messages. If that occurs, the write returns 0, and the 0-length message is discarded by the `ptem` module.

The standard *STREAMS* system calls can access the *pseudo-tty* devices. The slave devices support the `O_NDELAY` and `O_NONBLOCK` flags. Since the master side does not act like the terminal, if `O_NONBLOCK` or `O_NDELAY` is set, `read` on the master side returns with `errno` set to `[EAGAIN]` if no data are available, and `write` returns -1 with `errno` set to `[EAGAIN]` if there is internal flow control.

The master driver supports the `ISPTM` and `UNLKPT` `ioctl`s that are used by the functions `grantpt`, `unlockpt`, and `ptsname` [see `grantpt(3C)`, `unlockpt(3C)`, `ptsname(3C)`]. The `ioctl` `ISPTM` determines whether the file descriptor is that of an open master device. On success, it returns the major/minor number (type `dev_t`) of the master device which can be used to determine the name of the corresponding slave device. The `ioctl` `UNLKPT` unlocks the master and slave devices. It returns 0 on success. On failure, the `errno` is set to `[EINVAL]` indicating that the master device is not open.

The format of these commands is:

```
int ioctl(fd, command, arg)
    int fd, command, arg
```

where `command` is either `ISPTM` or `UNLKPT` and `arg` is 0. On failure, -1 is returned.

When data are written to the master side, the entire block of data written is treated as a single line. The slave side process reading the terminal receives the entire block of data. Data are not input edited by the `ldterm` module regardless of the terminal mode. The master side application is responsible for detecting an interrupt character and sending an interrupt signal `{SIGINT}` to the process in the slave side. This can be done as follows:

```
ioctl(fd, TIOCSIGNAL, SIGINT)
```

where `{SIGINT}` is defined in the file `'signal.h'`. When a process on the master side issues this `ioctl`, the argument is the number of the signal that should be sent. The specified signal is then sent to the process group on the slave side.

To summarize, the master driver and slave driver have the following characteristics:

- Each master driver has one-to-one relationship with a slave device based on major/minor device numbers.
- Only one open is allowed on a master device. Multiple opens are allowed on the slave device according to standard file mode and ownership permissions.
- Each slave driver minor device has a node in the file system.
- An open on a master device automatically locks out an open on the corresponding slave driver.
- A slave cannot be opened unless the corresponding master is open and has unlocked the slave.
- To provide a *tty* interface to the user, the *ldterm* and *ptem* modules are pushed on the slave side.
- A close on the master sends a hang-up to the slave and renders both *Streams* unusable after all data have been consumed by the process on the slave side.
- The last close on the slave side sends a 0-length message to the master but does not sever the connection between the master and slave drivers.

12.2.5.1 grantpt

The `grantpt` function changes the mode and the ownership of the slave device that is associated with the given master device. Given a file descriptor *fd*, `grantpt` first checks that the file descriptor is that of the master device. If so, it obtains the name of the associated slave device and sets the user id to that of the user running the process and the group id to *tty*. The mode of the slave device is set to 0620.

If the process is already running as root, the permission of the slave can be changed directly without invoking this function. The interface is:

```
grantpt(int fd);
```

The `grantpt` function returns 0 on success and -1 on failure. It fails if one or more of the following occurs: *fd* is not an open file descriptor, *fd* is not associated with a master device, the corresponding slave could not be accessed, or a system call failed because no more processes could be created.

12.2.5.2 unlockpt

The `unlockpt` function clears a lock flag associated with a master/slave device pair. Its interface is:

```
unlockpt(int fd);
```

The `unlockpt` returns 0 on success and -1 on failure. It fails if one or more of the following occurs: *fd* is not an open file descriptor or *fd* is not associated with a master device.

12.2.5.3 ptsname

The `ptsname` function returns the name of the slave device that is associated with the given master device. It first checks that the file descriptor is that of the master. If it is, it then determines the name of the corresponding slave device `/dev/pts/` and returns a pointer to a string containing the null-terminated path name. The return value points to static data whose content is overwritten by each call. The interface is:

```
char *ptsname(int fd)
```

The `ptsname` function returns a non-‘NULL’ path name upon success and a ‘NULL’ pointer upon failure. It fails if one or more of the following occurs: *fd* is not an open file descriptor or *fd* is not associated with the master device.

13 Reference

13.1 Files

13.1.1 User Header Files

STREAMS

`'stropts.h'`

User include file for the *STREAMS* options file. This file includes ioctl definitions for the *Stream Head*. This file includes `'sys/stropts.h'`. Files are organized this way for backward compatibility of header file locations.

`'sys/stropts.h'`

System include file for the *STREAMS* options file. This file includes ioctl definitions for the *Stream Head*. This file is included by `'stropts.h'`. Files are organized this way for backward compatibility of header file locations.

STREAMS logger

`'log.h'` User include file for the *STREAMS* logger.

`'sys/log.h'`

`'strlog.h'`

User include file for the *STREAMS* logger.

`'sys/strlog.h'`

STREAMS Administrative Driver

`'sad.h'` User include file for the *STREAMS Administrative Driver*.

`'sys/sc.h'`

`'sys/sad.h'`

13.1.2 System Header Files

STREAMS

`'sys/stream.h'`

`'sys/strsubr.h'`

`'sys/strconf.h'`

`'sys/strdebug.h'`

DDI/DKI

`'sys/debug.h'`

`'sys/kmem.h'`

`'sys/cmn_err.h'`

`'sys/dki.h'`

`'sys/ddi.h'`

DDI/DKI function declarations and defines for *Linux Fast-STREAMS*. Extension definitions will be included when one or more of `_LFS_SOURCE`, `_SVR4_SOURCE`, `_AIX_SOURCE`, `_HPUX_SOURCE`, `_OSF_SOURCE`, `_SUN_SOURCE`, `_LIS_SOURCE` or `_UW7_SOURCE` are defined.

`'sys/svr4ddi.h'`

SVR4 DDI compatibility function declarations and defines. This file should not be included directly, but will be included from `'sys/ddi.h'` when `_SVR4_SOURCE` is defined.

`'sys/aixddi.h'`

AIX DDI compatibility function declarations and defines. This file should not be included directly, but will be included from `'sys/ddi.h'` when `_AIX_SOURCE` is defined.

`'sys/hpuxddi.h'`

HP-UX DDI compatibility function declarations and defines. This file should not be included directly, but will be included from `'sys/ddi.h'` when `_HPUX_SOURCE` is defined.

`'sys/osfddi.h'`

OSF DDI compatibility function declarations and defines. This file should not be included directly, but will be included from `'sys/ddi.h'` when `_OSF_SOURCE` is defined.

`'sys/sunddi.h'`

Solaris DDI compatibility function declarations and defines. This file should not be included directly, but will be included from `'sys/ddi.h'` when `_SUN_SOURCE` is defined.

`'sys/lisddi.h'`

LiS DDI compatibility function declarations and defines. This file should not be included directly, but will be included from `'sys/ddi.h'` when `_LIS_SOURCE` is defined.

`'sys/uw7ddi.h'`

UnixWare DDI compatibility function declarations and defines. This file should not be included directly, but will be included from `'sys/ddi.h'` when `_UW7_SOURCE` is defined.

Miscellaneous

`'sys/spec_fs_i.h'`

`'sys/streams/config.h'`

13.2 Modules

13.2.1 Stream Head Module ("sth")

13.2.2 Connect Line Discipline Module ("connld")

13.2.3 Pipe Module ("pipemod")

13.2.4 STREAMS Configuration Module ("sc")

13.3 Drivers

13.3.1 Clone Driver ("clone")

13.3.2 Echo Driver ("echo")

13.3.3 FIFO Driver ("fifo")

13.3.4 Log Driver ("log")

13.3.5 Named STREAMS Device Driver ("nsdev")

13.3.6 Null STREAM Driver ("nuls")

13.3.7 Pipe Driver ("pipe")

13.3.8 STREAMS Administrative Driver ("sad")

13.4 System Calls

13.4.1 New System Calls

Just as experienced by the **Linux STREAMS** project, *Linux Fast-STREAMS* suffers from the inability to hook system calls that have been, otherwise, reserved in the **Linux** kernel for use by *STREAMS*. Therefore, these system calls are implemented as library calls in the 'libsttreams' library instead of as system calls in the 'libc' library.

getmsg(2)	getmsg(2) is not normally implemented as a system call: it is a library call that calls getpmsg(2).
getpmsg(2)	—
putmsg(2)	putmsg(2) is not normally implemented as a system call: it is a library call that calls putpmsg(2).
putpmsg(2)	—
fattach(2)	—
fdetach(2)	(Note: HP-UX mentions a fdetach system call: I do not know why.)
isastream(2)	—

13.4.2 Modifications to Old System Calls

<code>pipe(2)</code>	—
<code>open(2)</code>	—
<code>fcntl(2)</code>	—
<code>ioctl(2)</code>	—
<code>signal(2)</code>	—
<code>poll(2)</code>	—
<code>select(2)</code>	—
<code>read(2)</code>	The <code>read(2)</code> system call is overloaded with an invalid length to implement the <code>getmsg(2)</code> and <code>getpmsg(2)</code> system calls.
<code>readv(2)</code>	—
<code>write(2)</code>	The <code>write(2)</code> system call is overloaded with an invalid length to implement the <code>putmsg(2)</code> and <code>putpmsg(2)</code> system calls.
<code>writew(2)</code>	—
<code>close(2)</code>	—

13.5 Input-Output Controls

<code>I_NREAD</code>	—
<code>I_PUSH</code>	—
<code>I_POP</code>	—
<code>I_LOOK</code>	—
<code>I_FLUSH</code>	—
<code>I_SRDOPT</code>	—
<code>I_GRDOPT</code>	—
<code>I_STR</code>	—
<code>I_SETSIG</code>	—
<code>I_GETSIG</code>	—
<code>I_FIND</code>	—
<code>I_LINK</code>	—
<code>I_UNLINK</code>	—
<code>I_RECVFD</code>	non- <i>EFT</i> definition
<code>I_PEEK</code>	—
<code>I_FDINSERT</code>	—
<code>I_SENDFD</code>	—
<code>I_E_RECVFD</code>	<i>Unixware</i> : <i>EFT</i> definition
<code>I_SWROPT</code>	—
<code>I_GWROPT</code>	—
<code>I_LIST</code>	—
<code>I_PLINK</code>	—
<code>I_PUNLINK</code>	—
<code>I_FLUSHBAND</code>	—
<code>I_CKBAND</code>	—
<code>I_GETBAND</code>	—
<code>I_ATMARK</code>	—

I_SETCLTIME	—
I_GETCLTIME	—
I_CANPUT	—
I_SERROPT	<i>Solaris</i> only
I_GERROPT	<i>Solaris</i> only
I_ANCHOR	<i>Solaris</i> only
I_S_RECVFD	<i>UnixWare</i> only
I_STATS	<i>UnixWare</i> only
I_BIGPIPE	<i>UnixWare</i> only
I_GETTP	<i>UnixWare</i> only
I_AUTOPUSH	<i>Mac OT</i> only
I_HEAP_REPORT	<i>Mac OT</i> only
I_FIFO	<i>Mac OT</i> only
I_PUTPMSG	<i>LiS</i> and <i>Mac OT</i>
I_GETPMSG	<i>LiS</i> and <i>Mac OT</i>
I_FATTACH	<i>LiS</i> only
I_FDETACH	<i>LiS</i> only
I_PIPE	<i>LiS</i> and <i>Mac OT</i>

13.6 Module entry points

qopen(9)	Module queue open procedure.
put(9)	Module queue put procedure.
srv(9)	Module queue service procedure.
qadmin(9)	Module queue administrative procedure.
qclose(9)	Module queue close procedure.
timeout(9)	Timeout callback.
qtimeout(9)	Timeout callback.
bufcall(9)	Buffer callback.
qbufcall(9)	Buffer callback.
mi_bufcall(9)	Buffer callback.
freemsg(9)	Buffer free routine callout.

13.7 Structures

13.7.1 STREAMS Structures

Driver Structures

cdevsw(9)	Character device switch table.
devinfo(9)	Device information structure.

Module Structures

fmodsw(9)	Module switch table.
modinfo(9)	Module information structure.
module_info(9)	Module information structure.

`module_stat(9)` Module statistics structure.

Stream Structures

`streamtab(9)` Stream information table.
`streamadm(9)` Stream administration table.
`stdata(9)` Stream head data structure.
`shinfo(9)` Stream head information structure.

Queue Structures

`queue(9)` Message queue.
`qband(9)` Message queue band.
`bandinfo(9)` Message queue band information structure.
`qinit(9)` Message queue initialization.
`queinfo(9)` Message queue information structure.

Message Structures

`msgb(9)` Message block.
`datab(9)` Data block.
`mbinfo(9)` Message block information structure.
`dbinfo(9)` Data block information structure.

Ancilliary Structures

`strevent(9)` STREAMS event structure.
`seinfo(9)` STREAMS event information structure.
`strapush(9)` STREAMS autopush structure.
`apinfo(9)` STREAMS autopush information structure.

Additional Structures

`cred_t(9)` User credentials.

13.8 Registration

13.8.1 Linux Fast-STREAMS Registration

Registration

`register_clone(9)` Register a clone minor device number for a *STREAMS* driver.
`unregister_clone(9)` Unregister a clone minor device number for a *STREAMS* driver.

`register_cmajor(9)` Register file operations against a major device number.
`unregister_cmajor(9)` Unregister file operations against a major device number.
`register_strdev(9)` Register a *STREAMS* device against a device major number.
`unregister_strdev(9)` Unregister a *STREAMS* device against a device major number.

`register_strdrv(9)` Register a *STREAMS* driver.

<code>unregister_strdrv(9)</code>	Unregister a <i>STREAMS</i> driver.
<code>register_strmod(9)</code>	Register a <i>STREAMS</i> module.
<code>unregister_strmod(9)</code>	Unregister a <i>STREAMS</i> module.
<code>register_strnod(9)</code>	Register a <i>STREAMS</i> minor device node.
<code>unregister_strnod(9)</code>	Unregister a <i>STREAMS</i> minor device node.

Autopush

<code>autopush_add(9)</code>	Add an autopush list entry for a given <i>STREAMS</i> device number.
<code>autopush_del(9)</code>	Delete an autopush list entry for a given <i>STREAMS</i> device number.
<code>autopush_find(9)</code>	Find an autopush list entry for a given <i>STREAMS</i> device number.
<code>autopush_vml(9)</code>	Verify an autopush <i>STREAMS</i> module list.

Administration

<code>getadmin(9)</code>	Get the administration function pointer for a <i>STREAMS</i> module.
<code>getmid(9)</code>	Get the <i>STREAMS</i> module identifier by module name.

13.9 Message Handling

13.9.1 STREAMS Message Handling Utilities

<code>allocb(9)</code>	Allocate a <i>STREAMS</i> message and data block.
<code>esballoc(9)</code>	Allocate a <i>STREAMS</i> message and data block with a caller supplied data buffer.
<code>testb(9)</code>	Test if a <i>STREAMS</i> message can be allocated.
<code>bufcall(9)</code>	Install a <i>STREAMS</i> buffer callback.
<code>unbufcall(9)</code>	Remove a <i>STREAMS</i> buffer callback.
<code>copyb(9)</code>	Copy a <i>STREAMS</i> message block.
<code>copymsg(9)</code>	Copy a <i>STREAMS</i> message.
<code>dupb(9)</code>	Duplicate a <i>STREAMS</i> message block.
<code>dupmsg(9)</code>	Duplicate a <i>STREAMS</i> message.
<code>linkb(9)</code>	Link a message block into a <i>STREAMS</i> message.
<code>unlinkb(9)</code>	Unlink a message block from a <i>STREAMS</i> message.
<code>rmvb(9)</code>	Remove a message block from a <i>STREAMS</i> message.
<code>adjmsg(9)</code>	Trim bytes from the front or back of a <i>STREAMS</i> message.
<code>msgpullup(9)</code>	Pull up bytes in a <i>STREAMS</i> message.
<code>pullupmsg(9)</code>	Pull up the bytes in a <i>STREAMS</i> message.
<code>freeb(9)</code>	Frees a <i>STREAMS</i> message block.
<code>freemsg(9)</code>	Frees a <i>STREAMS</i> message.
<code>datamsg(9)</code>	Tests a <i>STREAMS</i> message type for data.
<code>msgdsize(9)</code>	Calculates the size of the data in a <i>STREAMS</i> message.

`pcmsg(9)` Test a *STREAMS* data block message type for priority control.

13.10 Queue Handling

13.10.1 UP Queue Handling Functions

<code>backq(9)</code>	Find the upstream or downstream <i>STREAMS</i> message queue.
<code>RD(9)</code>	Return the read queue of a <i>STREAMS</i> queue pair.
<code>WR(9)</code>	Return the write queue of a <i>STREAMS</i> queue pair.
<code>OTHERQ(9)</code>	Return the other queue of a <i>STREAMS</i> queue pair.
<code>SAMESTR(9)</code>	Test for <i>STREAMS</i> pipe or <i>FIFO</i> .
<code>qsize(9)</code>	Return the number of messages on a <i>STREAMS</i> message queue.
<code>bcanput(9)</code>	Test banded flow control on a <i>STREAMS</i> message queue.
<code>canput(9)</code>	Test flow control on a <i>STREAMS</i> message queue.
<code>qenable(9)</code>	Schedule a <i>STREAMS</i> message queue service routine.
<code>canenable(9)</code>	Test whether a <i>STREAMS</i> message queue can be scheduled.
<code>enableok(9)</code>	Allow a <i>STREAMS</i> message queue to be scheduled.
<code>noenable(9)</code>	Disable a <i>STREAMS</i> message queue from being scheduled.
<code>flushband(9)</code>	Flushes band <i>STREAMS</i> messages from a <i>STREAMS</i> message queue.
<code>flushq(9)</code>	Flushes messages from a <i>STREAMS</i> message queue.
<code>getq(9)</code>	Gets a message from a <i>STREAMS</i> message queue.
<code>insq(9)</code>	Insert a message into a <i>STREAMS</i> message queue.
<code>rmvq(9)</code>	Remove a message from a <i>STREAMS</i> message queue.
<code>qreply(9)</code>	Reply to a message from a <i>STREAMS</i> message queue.
<code>putq(9)</code>	Put a message onto a <i>STREAMS</i> message queue.
<code>putbq(9)</code>	Put a message back on a <i>STREAMS</i> message queue.
<code>putctl(9)</code>	Put a control message on a <i>STREAMS</i> message queue.
<code>putctl1(9)</code>	Put a 1 byte control message on a <i>STREAMS</i> message queue.

13.10.2 MP Queue Handling Functions

<code>bcanputnext(9)</code>	Test for banded flow control beyond a <i>STREAMS</i> message queue.
<code>canputnext(9)</code>	Test for flow control beyond a <i>STREAMS</i> message queue.
<code>put(9)</code>	Invoke the put procedure for a <i>STREAMS</i> module or driver with a <i>STREAMS</i> message.
<code>putnext(9)</code>	Put a message beyond a <i>STREAMS</i> message queue.
<code>putnextctl1(9)</code>	Put a one byte control message beyond a <i>STREAMS</i> message queue.
<code>putnextctl(9)</code>	Put a control message beyond a <i>STREAMS</i> message queue.
<code>qprocsoff(9)</code>	Disable a <i>STREAMS</i> message queue for multi-processing.
<code>qprocson(9)</code>	Enable a <i>STREAMS</i> message queue for multi-processing.
<code>freezestr(9)</code>	Freeze the state of a <i>STREAMS</i> message queue.

<code>unfreezestr(9)</code>	Thaw the state of a <i>STREAMS</i> message queue.
<code>strqget(9)</code>	Gets information about a <i>STREAMS</i> message queue. In the non- <i>MP</i> environment, it was typical to directly access the elements of the <i>queue</i> structure. In the <i>MP</i> environment, it is no longer safe to directly access elements of the <i>queue</i> structure in this fashion. The <code>strqget(9)</code> function provides the ability to retrieve information about <i>STREAMS</i> message queues in the <i>MP</i> environment.
<code>strqset(9)</code>	Sets attributes of a <i>STREAMS</i> message queue. In the non- <i>MP</i> environment, it was typical to directly access the elements of the <i>queue</i> structure. In the <i>MP</i> environment, it is no longer safe to directly access elements of the <i>queue</i> structure in this fashion. The <code>strqset(9)</code> function provides the ability to set attributes for <i>STREAMS</i> message queues in the <i>MP</i> environment.

13.11 Miscellaneous Functions

13.11.1 Miscellaneous DDI/DKI Functions

Memory Functions

<code>kmem_alloc(9)</code>	Allocate kernel memory.
<code>kmem_free(9)</code>	Deallocate kernel memory.
<code>kmem_zalloc(9)</code>	Allocate and zero kernel memory.
<code>kmem_fast_alloc(9)</code>	Allocate kernel memory, fast.
<code>kmem_fast_free(9)</code>	Deallocate kernel memory, fast.

Data Movement and Comparison

<code>bcopy(9)</code>	Copy byte strings.
<code>bzero(9)</code>	Zero a byte string.
<code>copyin(9)</code>	Copy user data in from user space to kernel space.
<code>copyout(9)</code>	Copy user data out to user space from kernel space.
<code>max(9)</code>	Determine the maximum of two integers.
<code>min(9)</code>	Determine the minimum of two integers.

Device Numbers

<code>getmajor(9)</code>	Get the internal major device number for a device.
<code>getminor(9)</code>	Get the internal minor device number for a device.
<code>makedevice(9)</code>	Create a device from major and minor device numbers.

Timers

<code>delay(9)</code>	Postpone the calling process for a number of clock ticks.
<code>timeout(9)</code>	Start a timer.
<code>untimeout(9)</code>	Stop a timer.

Time, Process and Privilege

<code>drv_getparm(9)</code>	Driver retrieval of kernel parameters.
<code>drv_hztomsec(9)</code>	Convert kernel ticks to milliseconds.
<code>drv_hztousec(9)</code>	Convert kernel ticks to microseconds.
<code>drv_msectohz(9)</code>	Convert milliseconds to kernel ticks.
<code>drv_priv(9)</code>	Check if the current process is privileged.
<code>drv_usectohz(9)</code>	Convert microseconds to kernel ticks.
<code>drv_usecwait(9)</code>	Delay for a number of microseconds.

Error Logging

<code>cmn_err(9)</code>	Print a kernel command error.
<code>strlog(9)</code>	Pass a message to the <i>STREAMS</i> logger.

File Manipulation

<code>mknod(9)</code>	Create a device node.
<code>mount(9)</code>	Mount a file system.
<code>umount(9)</code>	Unmount a file system.
<code>unlink(9)</code>	Unlink a file.

13.12 Extensions

There are a number of extensions to *SVR 4.2 MP STREAMS* that have been applied by implementations over the years. Some of these extensions are common enough across multiple implementations to be considered part of the *ipso facto* standard for *STREAMS*. **Linux Fast-STREAMS** implements these functions are part of the core set of *STREAMS* functions.

13.12.1 Common Extensions

<code>linkmsg(9)</code>	Link a message block to a <i>STREAMS</i> message.
<code>putctl2(9)</code>	Put a two byte control message on a <i>STREAMS</i> message queue.
<code>putnextctl2(9)</code>	Put a two byte control message on the next <i>STREAMS</i> message queue.
<code>weldq(9)</code>	Weld two (or four) <i>STREAMS</i> message queues together.
<code>unweldq(9)</code>	Unweld two (or four) <i>STREAMS</i> message queues.

13.12.2 Linux Fast-STREAMS Extensions

Internal Queue Functions

<code>allocq(9)</code>	Allocate a <i>STREAMS</i> queue pair.
<code>setq(9)</code>	Set sizes and procedures associated with a <i>STREAMS</i> message queue.
<code>qattach(9)</code>	Attach a module onto a <i>STREAMS</i> file.
<code>qopen(9)</code>	Call a <i>STREAMS</i> module or driver open routine.

qclose(9)	Closes a <i>STREAMS</i> module or driver.
qdetach(9)	Detach a module from a <i>STREAMS</i> file.
freeq(9)	Deallocate a <i>STREAMS</i> queue pair.

Flow Control

bcanget(9)	Test for message arrival on a band on a stream.
canget(9)	Test for message arrival on a stream.

13.12.3 Extensions from LiS 2.18.1

appq(9)	Append on <i>STREAMS</i> message after another.
esbcall(9)	Install a buffer callback for an extended <i>STREAMS</i> message block.
isdatablk(9)	Test a <i>STREAMS</i> data block for data type.
isdatamsg(9)	Test a <i>STREAMS</i> data block for data type.
kmem_zalloc_node(9)	(undoc).
msgsize(9)	Calculate the size of the message blocks in a <i>STREAMS</i> message.
qcountstrm(9)	Add all counts on all <i>STREAMS</i> message queues in a stream.
xmsgsize(9)	Calculate the size of message blocks in a <i>STREAMS</i> message.

13.13 Compatibility

13.13.1 SVR 4.2 MP DDI/DKI Compatibility Functions

SVR 4.2 MP Core Functions

lbolt(9) Time in ticks
since reboot.

SVR 4.2 MP Compatibility Module

itimerout(9)	Perform a timeout at an interrupt level.
major(9)	Get the internal major number of a device.
makedev(9)	Make a device number from internal major and minor device numbers.
minor(9)	Get the internal minor number of a device.
sleep(9)	Put a process to sleep.
vtop(9)	Convert virtual to physical address.
wakeup(9)	Wake a process.

Atomic Integers

SVR 4.2 MP provides a set of functions for manipulating atomic integers. The **Linux** kernel also has a set of equivalent functions for manipulating atomic integers. These functions are general purpose and not *STREAMS*-specific. For portability of *STREAMS* drivers and modules that utilize the *SVR 4.2 MP* functions, these functions have been added to the *SVR 4.2 MP* Compatibility Module.

ATOMIC_INT_ADD(9)	Add an integer value to an atomic integer.
ATOMIC_INT_ALLOC(9)	Allocate and initialize an atomic integer.
ATOMIC_INT_DEALLOC(9)	Deallocate an atomic integer.
ATOMIC_INT_DECR(9)	Decrement and test an atomic integer.
ATOMIC_INT_INCR(9)	Increment an atomic integer.
ATOMIC_INT_INIT(9)	Initialize an atomic integer.
ATOMIC_INT_READ(9)	Read an atomic integer.
ATOMIC_INT_SUB(9)	Subtract an integer value from an atomic integer.
ATOMIC_INT_WRITE(9)	Write an integer value to an atomic integer.

Basic Locks

SVR 4.2 MP provides a set of functions for manipulating basic (spin) locks. The **Linux** kernel also has a set of equivalent functions for manipulating spin locks. These functions are general purpose and not *STREAMS*-specific. For portability of *STREAMS* drivers and modules that utilize the *SVR 4.2 MP* functions, these functions have been added to the *SVR 4.2 MP* Compatibility Module.

LOCK(9)	Lock a basic lock.
LOCK_ALLOC(9)	Allocate a basic lock.
LOCK_DEALLOC(9)	Deallocate a basic lock.
LOCK_OWNED(9)	Determine whether a basic lock is held by the caller.
TRYLOCK(9)	Try to lock a basic lock.
UNLOCK(9)	Unlock a basic lock.

STREAMS Locks

SVR 4.2 MP defines a set of *STREAMS*-specific locks. The **Linux** kernel does not provide these functions. *Linux Fast-STREAMS* has some equivalent internal functions. For portability of *STREAMS* drivers and modules that utilize the *SVR 4.2 MP* functions, these functions have been added to the *SVR 4.2 MP* Compatibility Module.

MPSTR_QLOCK(9)	Release a queue from exclusive access.
MPSTR_QRELE(9)	Acquire a queue for exclusive access.
MPSTR_STPLOCK(9)	Acquire a stream head for exclusive access.
MPSTR_STPRELE(9)	Release a stream head from exclusive access.

Read/Write Locks

SVR 4.2 MP provides a set of functions for manipulating read-write locks. The **Linux** kernel also has a set of equivalent functions for manipulating spin locks. These functions are general purpose and not *STREAMS*-specific. For portability of *STREAMS* drivers and modules that utilize the *SVR 4.2 MP* functions, these functions have been added to the *SVR 4.2 MP* Compatibility Module.

RW_ALLOC(9)	Allocate and initialize a read/write lock.
RW_DEALLOC(9)	Deallocate a read/write lock.
RW_RDLOCK(9)	Acquire a read/write lock in read mode.
RW_TRYRDLOCK(9)	Attempt to acquire a read/write lock in read mode.
RW_TRYWRLOCK(9)	Attempt to acquire a read/write lock in write mode.

RW_UNLOCK(9)	Release a read/write lock.
RW_WRLOCK(9)	Acquire a read/write lock in write mode.

Priority Levels

sp10(9)	Set priority level 0.
sp11(9)	Set priority level 1.
sp12(9)	Set priority level 2.
sp13(9)	Set priority level 3.
sp14(9)	Set priority level 4.
sp15(9)	Set priority level 5.
sp17(9)	Set priority level 6.
sp17(9)	Set priority level 7.
spl(9)	Set priority level.
splx(9)	Set priority level x.

Sleep Locks

SVR 4.2 MP provides a set of functions for manipulating sleep locks. The **Linux** kernel also has a set of equivalent functions for manipulating semaphores. These functions are general purpose and not *STREAMS*-specific. For portability of *STREAMS* drivers and modules that utilize the *SVR 4.2 MP* functions, these functions have been added to the *SVR 4.2 MP* Compatibility Module.

SLEEP_ALLOC(9)	Allocate a sleep lock.
SLEEP_DEALLOC(9)	Deallocate a sleep lock.
SLEEP_LOCK(9)	Acquire a sleep lock.
SLEEP_LOCKAVAIL(9)	Determine whether a sleep lock is available.
SLEEP_LOCKOWNED(9)	Determine whether a sleep lock is held by the caller.
SLEEP_LOCK_SIG(9)	Acquire a sleep lock.
SLEEP_TRYLOCK(9)	Attempt to acquire a sleep lock.
SLEEP_UNLOCK(9)	Release a sleep lock.

Synchronization Variables

SVR 4.2 MP provides a set of functions for manipulating synchronization variables. The **Linux** kernel also has a set of equivalent functions for manipulating wait queues. These functions are general purpose and not *STREAMS*-specific. For portability of *STREAMS* drivers and modules that utilize the *SVR 4.2 MP* functions, these functions have been added to the *SVR 4.2 MP* Compatibility Module.

SV_ALLOC(9)	Allocate a basic condition variable.
SV_BROADCAST(9)	Broadcast a basic condition variable.
SV_DEALLOC(9)	Deallocate a basic condition variable.
SV_SIGNAL(9)	Signal a basic condition variable.
SV_WAIT(9)	Wait on a basic condition variable.
SV_WAIT_SIG(9)	Interruptible wait on a basic condition variable.

Resource Allocation

<code>rmalloc(9)</code>	Allocate a number of units from a resource map.
<code>rmapalloc(9)</code>	Allocated a resource map.
<code>rmapalloc_wait(9)</code>	Allocated a resource map.
<code>rmapalloc_wait(9)</code>	Allocate a number of units from a resource map.
<code>rmfree(9)</code>	Free a number of units from a resource map.
<code>rmfreemap(9)</code>	Free a resource map.
<code>rmget(9)</code>	Allocated a number of units from a resource map.
<code>rminit(9)</code>	Initialize a resource map.
<code>rmsetwant(9)</code>	Wait for resources on a resource map.
<code>rmwanted(9)</code>	Waits on a resource map.

13.13.2 AIX 5L Version 5.1 Compatibility Functions

The following functions are provided by *Linux Fast-STREAMS* for compatibility with the *AIX 5L Version 5.1 Portable STREAMS Environment (PSE)*:

AIX Core Functions

The functions in this section are provided as part of the core functions provided in the **Linux Fast-STREAMS** *STREAMS* subsystem:

<code>putctl2(9)</code>	Put a 2 byte control message on a <i>STREAMS</i> message queue. Many <i>STREAMS</i> implementations provide this function. When the errors that can be delivered to the Stream Head in a <code>M_ERROR</code> message were broken from a single read/write error condition to a separate read and write error condition, the <code>putctl(9)</code> function lost much of its utility. Implementation of a <code>putctl2(9)</code> function provides a similar capability for read/write error conditions as <code>putctl(9)</code> provided for combined error conditions previously. <i>AIX</i> implements this function. <i>Linux Fast-STREAMS</i> provides this function for compatibility with <i>AIX</i> and other <i>SVR 4.2 MP</i> based <i>STREAMS</i> implementations. <code>putctl2(9)</code> is not very useful in an <i>MP</i> environment, where one really wants to place <code>M_ERROR</code> messages on the <i>upstream</i> queue. The function for that in the <i>MP</i> environment is <code>putnextctl2(9)</code> . <i>AIX</i> does not implement the <code>putnextctl2(9)</code> function, which is somewhat surprising, but then <i>AIX</i> does not really handle <i>MP</i> environments the same way that <i>SVR 4.2 MP</i> does: the <i>AIX Portable STREAMS Environment (PSE)</i> is essentially single threaded, and <i>AIX PSE</i> does not implement <code>QHLIST</code> meaning that the synchronization for protecting dereferencing of ‘ <code>q->q_next</code> ’ pointers is not present.
-------------------------	--

<code>splstr(9)</code>	Set or restore priority levels. Although <i>Linux Fast-STREAMS</i> does not interpret priority levels in the same fashion as <i>SVR 4.2 MP</i> , the <code>splstr(9)</code> is provided in support of <i>AIX</i> and other <i>STREAMS</i> implementations. <i>AIX</i> also does not interpret priority levels in the same manner, but provides <code>splstr(9)</code> and <code>splx(9)</code> functions in support of <i>STREAMS</i> .
<code>splx(9)</code>	Set or restore priority levels. Although <i>Linux Fast-STREAMS</i> does not interpret priority levels in the same fashion as <i>SVR 4.2 MP</i> , the <code>splx(9)</code> is provided in support of <i>AIX</i> and other <i>STREAMS</i> implementations. <i>AIX</i> also does not interpret priority levels in the same manner, but provides <code>splstr(9)</code> and <code>splx(9)</code> functions in support of <i>STREAMS</i> .
<code>unweldq(9)</code> <code>weldq(9)</code>	Unweld two pairs of <i>STREAMS</i> message queues. Weld together two pairs of <i>STREAMS</i> message queues.

AIX Compatibility Module

The functions in this section are provided as part of the *AIX Compatibility Module* contained in the ‘`streams-aixcompat.o`’ kernel module.

<code>mi_bufcall(9)</code>	Reliable alternative to <code>bufcall(9)</code> . <code>mi_bufcall(9)</code> is an <i>AIX</i> -specific function. <i>Linux Fast-STREAMS</i> provides this function in support of <i>AIX</i> drivers and modules. This function will not be used by portable <i>STREAMS</i> modules and drivers intended to be portable across <i>STREAM</i> implementations based on <i>SVR 4.2 MP</i> .
<code>mi_close_comm(9)</code>	<i>STREAMS</i> common minor device close utility. <code>mi_close_comm(9)</code> is an <i>AIX</i> -specific function. <i>Linux Fast-STREAMS</i> provides this function in support of <i>AIX</i> drivers and modules. This function will not be used by portable <i>STREAMS</i> modules and drivers intended to be portable across <i>STREAM</i> implementations based on <i>SVR 4.2 MP</i> .
<code>mi_next_ptr(9)</code>	<i>STREAMS</i> minor device list traversal. <code>mi_next_ptr(9)</code> is an <i>AIX</i> -specific function. <i>Linux Fast-STREAMS</i> provides this function in support of <i>AIX</i> drivers and modules. This function will not be used by portable <i>STREAMS</i> modules and drivers intended to be portable across <i>STREAM</i> implementations based on <i>SVR 4.2 MP</i> .

<code>mi_open_comm(9)</code>	<i>STREAMS</i> common minor device open utility. <code>mi_open_comm(9)</code> is an <i>AIX</i> -specific function. <i>Linux Fast-STREAMS</i> provides this function in support of <i>AIX</i> drivers and modules. This function will not be used by portable <i>STREAMS</i> modules and drivers intended to be portable across <i>STREAM</i> implementations based on <i>SVR 4.2 MP</i> .
<code>mi_prev_ptr(9)</code>	<i>STREAMS</i> minor device list traversal. <code>mi_prev_ptr(9)</code> is an <i>AIX</i> -specific function. <i>Linux Fast-STREAMS</i> provides this function in support of <i>AIX</i> drivers and modules. This function will not be used by portable <i>STREAMS</i> modules and drivers intended to be portable across <i>STREAM</i> implementations based on <i>SVR 4.2 MP</i> .
<code>str_install(9)</code>	Install a <i>STREAMS</i> module or driver. <code>str_install(9)</code> is the <i>AIX</i> -specific driver and module registration and deregistration facility. This facility is fashioned after the <i>SVR 4.2 MP</i> facility. <i>Linux Fast-STREAMS</i> provides an <i>AIX</i> version of this function in support of <i>AIX</i> . Only the <i>SVR 4.2 MP</i> version of this function will be used by portable <i>STREAMS</i> modules and drivers intended to be portable across <i>STREAM</i> implementations based on <i>SVR 4.2 MP</i> .
<code>wantio(9)</code>	Perform direct I/O from a <i>STREAMS</i> driver. <code>wantio(9)</code> is an <i>AIX</i> -specific function. <i>Linux Fast-STREAMS</i> provides this function in support of <i>AIX</i> drivers and modules. This function will not be used by portable <i>STREAMS</i> modules and drivers intended to be portable across <i>STREAM</i> implementations based on <i>SVR 4.2 MP</i> .
<code>wantmsg(9)</code>	Provide a filter of wanted messages from a <i>STREAMS</i> module. <code>wantmsg(9)</code> is an <i>AIX</i> -specific function. <i>Linux Fast-STREAMS</i> provides this function in support of <i>AIX</i> drivers and modules. This function will not be used by portable <i>STREAMS</i> modules and drivers intended to be portable across <i>STREAM</i> implementations based on <i>SVR 4.2 MP</i> .

13.13.3 HP-UX 11.0i v2 Compatibility Functions

The following functions are provided by *Linux Fast-STREAMS* for compatibility with *HP-UX 11.0i v2*:

***HP-UX* Core Functions**

The following functions are provided by *Linux Fast-STREAMS* for compatibility with the *HP-UX 11.0i v2 STREAMS/UX*:

<code>putctl2(9)</code>	Put a 2 byte control message on a <i>STREAMS</i> message queue. Many <i>STREAMS</i> implementations provide this function. When the errors that can be delivered to the Stream Head in a <code>M_ERROR</code> message were broken from a single read/write error condition to a separate read and write error condition, the <code>putctl(9)</code> function lost much of its utility. Implementation of a <code>putctl2(9)</code> function provides a similar capability for read/write error conditions as <code>putctl(9)</code> provided for combined error conditions previously. <i>HP-UX</i> implements this function. <i>Linux Fast-STREAMS</i> provides this function for compatibility with <i>HP-UX</i> and other <i>SVR 4.2 MP</i> based <i>STREAMS</i> implementations.
<code>putnextctl2(9)</code>	Put a 2 byte control message on the downstream <i>STREAMS</i> message queue. Many <i>STREAMS MP</i> implementations provide this function. When the errors that can be delivered to the Stream Head in a <code>M_ERROR</code> message were broken from a single read/write error condition to a separate read and write error condition, the <code>putnextctl(9)</code> function lost much of its utility. Implementation of a <code>putnextctl2(9)</code> function provides a similar capability for read/write error conditions as <code>putnextctl(9)</code> provided for combined error conditions previously. <i>HP-UX</i> implements this function. <i>Linux Fast-STREAMS</i> provides this function for compatibility with <i>HP-UX</i> and other <i>SVR 4.2 MP</i> based <i>STREAMS</i> implementations.
<code>unweldq(9)</code> <code>weldq(9)</code>	Unweld two pairs of <i>STREAMS</i> message queues. Weld together two pairs of <i>STREAMS</i> message queues.

HP-UX Compatibility Module

The functions in this section are provided as part of the *HP-UX Compatibility Module* contained in the ‘`streams-hpuxcompat.o`’ kernel module.

<code>str_install(9)</code>	Install a <i>STREAMS</i> module or driver. <code>str_install(9)</code> is the <i>HP-UX</i> -specific driver and module registration facility. This facility is fashioned after the <i>SVR 4.2 MP</i> facility. <i>Linux Fast-STREAMS</i> provides an <i>HP-UX</i> version of this function in support of <i>HP-UX</i> . Only the <i>SVR 4.2 MP</i> version of this function will be used by portable <i>STREAMS</i> modules and drivers intended to be portable across <i>STREAM</i> implementations based on <i>SVR 4.2 MP</i> .
-----------------------------	---

<code>str_uninstall(9)</code>	Uninstall a <i>STREAMS</i> module or driver. <code>str_uninstall(9)</code> is the <i>HP-UX</i> -specific driver and module deregistration facility. This facility is fashioned after the <i>SVR 4.2 MP</i> facility. <i>Linux Fast-STREAMS</i> provides an <i>HP-UX</i> version of this function in support of <i>HP-UX</i> . Only the <i>SVR 4.2 MP</i> version of this function will be used by portable <i>STREAMS</i> modules and drivers intended to be portable across <i>STREAM</i> implementations based on <i>SVR 4.2 MP</i> .
<code>streams_get_sleep_lock(9)</code>	Provide access to the global sleep lock.
<code>streams_put(9)</code>	Invoke the put procedure for a <i>STREAMS</i> module or driver with a <i>STREAMS</i> message.

13.13.4 OSF/1 1.2/Digital UNIX Compatibility Functions

OSF/1 Core Functions

The following functions are provided by *Linux Fast-STREAMS* for compatibility with the *OSF/1 1.2/Digital UNIX*:

<code>unweldq(9)</code>	Unweld two pairs of <i>STREAMS</i> message queues.
<code>weldq(9)</code>	Weld together two pairs of <i>STREAMS</i> message queues.

OSF/1 Compatibility Module

The functions in this section are provided as part of the *OSF/1 Compatibility Module* contained in the ‘`streams-osfcompat.o`’ kernel module.

<code>lbolt(9)</code>	Time in ticks since reboot.
<code>puthere(9)</code>	Invoke the put procedure for a <i>STREAMS</i> module or driver with a <i>STREAMS</i> message.
<code>streams_close_comm(9)</code>	Common minor device close utility.
<code>streams_open_comm(9)</code>	Common minor device open utility.
<code>streams_open_ocomm(9)</code>	Common minor device open utility.
<code>strmod_add(9)</code>	Add a <i>STREAMS</i> module.
<code>strmod_del(9)</code>	Delete a <i>STREAMS</i> module or driver from the kernel.
<code>time(9)</code>	(undoc).

13.13.5 UnixWare 7.1.3 (OpenUnix 8) Compatibility Functions

UnixWare Core Functions

The following functions are provided by *Linux Fast-STREAMS* for compatibility with the *UnixWare 7.1.3 (OpenUnix 8)*:

UnixWare Compatibility Module

The functions in this section are provided as part of the *UnixWare Compatibility Module* contained in the ‘streams-uw7compat.o’ kernel module.

allocb_physreq(9)	Allocate a <i>STREAMS</i> message and data block.
emajor(9)	Get the external (real) major device number from the device number.
eminor(9)	Get the external extended minor device number from the device number.
etoimajor(9)	Convert an external major device number to an internal major device number.
getemajor(9)	Get the external (real) major device number.
geteminor(9)	Get the external minor device number.
itoemajor(9)	Convert an internal major device number to an external major device number.
msgphysreq(9)	Cause a message block to meet physical requirements.
msgpullup_physreq(9)	Pull up bytes in a <i>STREAMS</i> message.
msgscgth(9)	(undoc).
printf(9)	(undoc).
striocall(9)	(undoc).

13.13.6 Solaris 9/SunOS 5.9 Compatibility Functions

Solaris Core Functions

The following functions are provided by *Linux Fast-STREAMS* for compatibility with the *Solaris 9/SunOS 5.9*:

Solaris Compatibility Module

The functions in this section are provided as part of the *Solaris Compatibility Module* contained in the ‘streams-suncompat.o’ kernel module.

_fini(9)	
_info(9)	
_init(9)	
attach(9)	attach a device to the system or resume a suspended device
ddi_create_minor_node(9)	create a minor node for this device
ddi_driver_major(9)	find the major device number associated with a driver
ddi_driver_name(9)	return normalized driver name
ddi_get_cred(9)	get a reference to the credentials of the current user
ddi_getiminor(9)	
ddi_get_instance(9)	get device instance number
ddi_get_lbolt(9)	get the current value of the system tick clock

<code>ddi_get_pid(9)</code>	
<code>ddi_get_soft_state(9)</code>	
<code>ddi_get_time(9)</code>	get the current time in seconds since the epoch
<code>ddi_remove_minor_node(9)</code>	remove a minor node for a device
<code>ddi_removing_power(9)</code>	
<code>ddi_soft_state(9)</code>	
<code>ddi_soft_state_fini(9)</code>	
<code>ddi_soft_state_free(9)</code>	
<code>ddi_soft_state_init(9)</code>	
<code>ddi_soft_state_zalloc(9)</code>	
<code>ddi_umem_alloc(9)</code>	allocate page aligned kernel memory
<code>ddi_umem_free(9)</code>	
<code>detach(9)</code>	detach a device from the system or suspend a device
<code>getinfo(9)</code>	
<code>identify(9)</code>	determine if a driver is associated with a device
<code>install_driver(9)</code>	install a device driver
<code>mod_info(9)</code>	provides information on a loadable kernel module to the <i>STREAMS</i> executive
<code>mod_install(9)</code>	installs a loadable kernel module in the <i>STREAMS</i> executive
<code>mod_remove(9)</code>	removes a loadable module from the <i>STREAMS</i> executive
<code>power(9)</code>	power a device attached to the system
<code>probe(9)</code>	
<code>qbufcall(9)</code>	install a buffer callback
<code>qtimeout(9)</code>	start a timer associated with a queue
<code>queclass(9)</code>	
<code>qunbufcall(9)</code>	Cancel a <i>STREAMS</i> buffer callback.
<code>quntimeout(9)</code>	
<code>qwait(9)</code>	wait for a queue message
<code>qwait_sig(9)</code>	wait for a queue message or signal
<code>qwriter(9)</code>	

13.13.7 LiS 2.18.1 Compatibility Functions

```

lis_register_strdev(9)
lis_register_strmod(9)
lis_unregister_strdev(9)
lis_unregister_strmod(9)

lis_adjmsg(9)
lis_allocb(9)
lis_allocb_physreq(9)
lis_allocq(9)
lis_appq(9)
lis_backq(9)

```

lis_bcanput(9)
lis_bcanputnext(9)
lis_bcopy(9)
lis_bufcall(9)
lis_bzero(9)
lis_canenable(9)
lis_canput(9)
lis_canputnext(9)
lis_cmn_err(9)
lis_copyb(9)
lis_copymsg(9)
lis_datamsg(9)
lis_date(9)
lis_dupb(9)
lis_dupmsg(9)
lis_enableok(9)
lis_esballoc(9)
lis_esbbc(9)
lis_find_strdev(9)
lis_flushband(9)
lis_flushq(9)
lis_freeb(9)
lis_freemsg(9)
lis_freeq(9)
lis_getq(9)
lis_insq(9)
lis_isdatabl(9)
lis_isdatamsg(9)
lis_linkb(9)
lis_mknod(9)
lis_mount(9)
lis_msgdsize(9)
lis_msgpullup(9)
lis_msgsize(9)
lis_noenable(9)
lis_OTHER(9)
lis_OTHERQ(9)
lis_pullupmsg(9)
lis_putbq(9)
lis_putctl1(9)
lis_putctl(9)
lis_putnext(9)
lis_putnextctl1(9)
lis_putnextctl(9)
lis_putq(9)

lis_qattach(9)
lis_qclose(9)
lis_qdetach(9)
lis_qenable(9)
lis_qopen(9)
lis_qprocsoff(9)
lis_qprocson(9)
lis_qreply(9)
lis_qsize(9)
lis_RD(9)
lis_register_strdev(9)
lis_register_strmod(9)
lis_rmvb(9)
lis_rmvq(9)
lis_safe_canenable(9)
lis_safe_enableok(9)
lis_safe_noenable(9)
lis_safe_OTHERQ(9)
lis_safe_putnext(9)
lis_safe_qreply(9)
lis_safe_RD(9)
lis_safe_SAMESTR(9)
lis_safe_WR(9)
lis_SAMESTR(9)
lis_stream_utils(9)
lis_strqget(9)
lis_strqset(9)
lis_testb(9)
lis_timeout(9)
lis_umount2(9)
lis_umount(9)
lis_unbufcall(9)
lis_unlink(9)
lis_unlinkb(9)
lis_unregister_strdev(9)
lis_unregister_strmod(9)
lis_untimeout(9)
lis_version(9)
lis_WR(9)
lis_xmsgsize(9)

14 Examples

14.1 Module Example

14.2 Driver Example

15 Device Numbers

Linux Fast-STREAMS supports the concept of internal and external device numbering with base majors and extended minors.

Linux Fast-STREAMS uses several device numbering schemes intended to be compatible with *SVR 4.2 MP DDI/DKI* and implementations based on *SVR 4.2* including AIX, HP-UX, LiS, OSF/1, Solaris, Super/UX, UnixWare, and UXP/V.

15.1 External Device Numbers

Linux Fast-STREAMS provides real external device numbering using the functions `getemajor(9)`, `geteminor(9)` and `makedevice(9)`. When used on a *devp* argument passed to the `qopen(9)` procedures of a *STREAMS* driver or module, `getemajor(9)` and `geteminor(9)` will return the real external major or minor device number. The real external major or minor device number is the major or minor device number that is seen by user applications.

A number of administrative utilities are provided that assist with the assignment of device names and number and are useful in boot scripts: (see [Section 17.1 \[Administrative Utilities\]](#), page 255)

<code>autopush(8)</code>	– verify modules and establish autopush lists
<code>insf(8)</code>	– install special device files
<code>scls(8)</code>	– list <i>STREAMS</i> drivers
<code>strinfo(8)</code>	– provide information on <i>STREAMS</i> drivers and modules
<code>strload(8)</code>	– load or unload <i>STREAMS</i> drivers and modules
<code>strsetup(8)</code>	– create <i>STREAMS</i> devices

Another approach to establishment of dynamically allocated major device numbers is to use the *LiS* approach of creating minor device nodes using the `mknod(9)` and `unlink(9)` facilities provided by the *LiS* compatibility module. But this approach does not work well with demand loading of kernel modules because it relies upon the **Linux** character device demand loading approach, or requires informing *STREAMS* of all devices and drivers when *STREAMS* is being compiled.

Linux Fast-STREAMS supports the **Linux** character device module demand loading; however, the **Linux** mechanism requires prior knowledge of the character major device number. Some of the administrative utilities above can load kernel modules and establish what major device number was assigned. See the individual manual pages for more information.

15.2 Internal Device Numbers

Internal extended minor device number is a scheme whereby the *STREAMS* device driver does not have to be concerned with architectural limitations on the number of minor device number available to user applications (0 to 255 in **Linux 2.4**). Instead, the driver treats the major device number as a base internal device number against which a range of minor devices can be assigned (currently 0 to 65535). For external applications, device numbering consists of multiple (real) external major device numbers that correspond to a single base

internal major device number. Internal extended device numbering is provided by the `getmajor(9)`, `getminor(9)` and `makedevice(9)` facilities. Conversions between external (real) major device numbers and internal (base) major device numbers can be performed with the `etoimajor(9)` and `itoemajor(9)` utilities provided by the *UnixWare* compatibility modules.

Extended device numbering further complicates the matter of dynamic assignment of major device numbers and makes it harder for boot and configuration scripts to properly create device nodes.

15.3 Clone Device

15.3.1 Traditional Cloning

The `clone(4)` driver supports traditional clone devices. Traditional clone devices work by assigning the clone major device number and a unique minor device number to a “clone” device. When this device is opened, the minor device number of the device is used as the new major device number and zero (0) is used as the minor device number, and the `qopen(9)` is chained to the `qopen(9)` procedure of the new `streamtab(9)` structure resulting from looking up this newly created device number. *sflag* is always set to ‘CLONEOPEN’ when the traditional clone driver opens a device.

`getemajor(9)`, `geteminor(9)`, `getmajor(9)` and `getminor(9)` still function as normal on the resulting *devp* argument passed to the ultimate driver’s `qopen(9)` procedure. This is true both for base and extended major device numbers. The following table illustrates the situation (assuming that the major device number assigned to the `clone(4)` device is 72):

clone	external	internal	sflag
72:32	32:0	32:000	‘CLONEOPEN’
72:35	35:0	32:256	‘CLONEOPEN’
72:37	37:0	32:512	‘CLONEOPEN’

If a driver wishes to assign an extended major device number in response to a **CLONEOPEN**, it should use `makedevice(9)` with the major device number obtained with `getmajor(9)` and an extended minor device number assigned by the driver. In this way, extended minor device numbers are transparent to the operation of the `clone(4)` driver and ‘CLONEOPEN’.

15.3.2 New Cloning

Linux Fast-STREAMS supports the new cloning approach whereby a driver is permitted to alter the device number returned in the *devp* argument to `qopen(9)` even though *sflag* is set to ‘DRVOPEN’ instead of ‘CLONEOPEN’. The driver, at its discretion, can treat any minor device number as a new style clone device, although normally minor device number zero (0) is usually used as the clone minor.

The driver may either use an external (real) major device number or the internal (base) major device number. That is, when calling `makedevice(9)` to create the device to return to the *devp* argument to `qopen(9)`, the driver can use an external (real) major device number (returned by `getemajor(9)`) combined with an assigned external (real) minor device number (from 0 to 255); or, it can use an internal (base) major device number (returned by

`getmajor(9)`) combined with an assigned extended minor device number (from 0 to 65535). The latter approach is often easier to use.

15.4 Named STREAMS Device

The `nsdev(4)` driver provides for a *Named STREAMS Device*. This is a device numbering approach unique to *Linux Fast-STREAMS*. The `nsdev(4)` driver operates in a similar manner to the `clone(4)` driver, however, the major number is derived from the prefix name of the device and the minor number is taken directly from the minor number of the `nsdev(4)` device.

The following table summarizes the approach (assuming that the major device numbers assigned to the `nsdev(4)` device are 74, 75, 76 and 77 and the major device numbers assigned to the `inet(4)` driver are 32, 35 and 37):

nsdev		external	internal	<i>sflag</i>
inet	74:000	32:000	32:000	'DRVOPEN'
inet.udp	74:017	32:017	32:017	'DRVOPEN'
inet.99	74:099	32:099	32:099	'DRVOPEN'
inet.512	76:000	37:000	32:512	'DRVOPEN'

The `nsdev(4)` device also has the characteristic that if a device name with a major device number of the `nsdev(4)` device is opened and there is no device loaded that corresponds to the prefix name of the device from which to obtain a major device number, the `nsdev(4)` will attempt to load the '**streams-prefix**' kernel module using `request_module(9)` and try again: where, '**prefix**' is the prefix of the device name up to the first point character (corresponding to a digit `.`). So, in the example above, if the '`inet`' driver was not loaded, but an attempt was made to open the `/dev/inet.99` device, *STREAMS* would request the '**streams-inet**' kernel module be loaded. This approach simplifies kernel module loading as well as device numbering and makes it easier for boot scripts to initialize devices.

15.5 spec File System

Another approach to creation and assignment of device numbers is the mountable `specfs(5)` file system. The `specfs(5)` file system can be mounted to provide an in-kernel device directory similar to the **Linux** '`devfs`' file system and the *Solaris* devices file system. The `specfs(5)` file system should normally be mounted on the `/dev/streams` subdirectory by system initialization scripts using a command such as: `mount -t specfs none /dev/streams`. See `mount(8)` for more information. Once mounted over the `/dev/streams` subdirectory, subdirectories of `/dev/streams` corresponding to each loaded driver will appear. So, for example, when the `inet(4)` driver is loaded, the "inet" subdirectory will appear at `/dev/streams/inet`. Within each device subdirectory `/dev/streams/devicename/`, each instance of the device will appear as a character device named with the instance number of the device and having the eternal (real) major and external (real) minor device number.

The mounted `specfs(5)` file system also has the characteristic that if a device subdirectory `/dev/streams/devname/` does not exist, but an attempt is made to read such a directory, *Linux Fast-STREAMS* will attempt to load kernel module '**streams-devname**' into the

kernel with `request_module(9)`. If the load is successful, the kernel module will register and the subdirectory will be created and read.

Also, if an attempt is made to open a numbered file within a device subdirectory of `‘/dev/streams/devname/nnnn’` where, `‘nnnn’` is an octal, hexadecimal or decimal *ASCII* number, *STREAMS* will open the driver (call `qopen(9)` for driver `‘devname’`) with the instance number resulting from the conversion of the device name `‘nnnn’` to an instance number.

These two characteristics permit symbolic links to be placed in the `‘/dev’` directory that link to a device name and instance number in the `‘/dev/stream’` directory. An example is given in the table below:

link	external	internal	sflag
<code>‘/dev/tcp’ -> ‘/dev/streams/inet/36’</code>	30:36	30:36	‘DRVOPEN’
<code>‘/dev/udp’ -> ‘/dev/streams/inet/39’</code>	30:39	30:39	‘DRVOPEN’

In fact, it does not matter what the character device major or minor device number is on the node in the `specfs(5)` file system. This is because the inode in the file system is directly associated with the `streamtab(9)` structure and instance number without using the normal **Linux** character device mechanisms. When a device instance exceeds the extended minor device numbering space assigned to a device driver in the `specfs(5)` file system, device number displayed by `stat(2)`, `lstat(2)` or `fstat(2)` is chosen by wrapping the instance number into the extended minor device numbering space.

This approach makes it unnecessary to statically assign major device numbers, or to dynamically assign major device numbers to devices in boot scripts, and is by far the easiest approach. All that is required by packages at installation is that they establish the necessary symbolic links on device name and instance number without concern for major device numbers.

16 Multi-Threading

The **Linux** 2.6 kernel is multi-threaded to make effective use of symmetric shared-memory multiprocessor computers. All parts of the kernel, including *STREAMS* modules and drivers, must ensure data integrity in a multiprocessing environment. For the most part, developers must ensure that concurrently running kernel threads do not attempt to manipulate the same data at the same time. The *STREAMS* framework provides multiprocessing *Synchronization Levels*, which allows the developer control over the level of concurrency allowed in a module. The *SVR 4.2 MP DDI/DKI* also provides locking mechanisms for protecting data.

There are two types of entry points, callbacks and callouts in the *Linux Fast-STREAMS* subsystem:

1. *Synchronous*. These entry points, callbacks and callouts are referenced against a *STREAMS* queue structure. That is, they were invoked using a *STREAMS* queue structure as an argument. These procedures are as follows:

put(9)	—
srv(9)	—
qopen(9)	—
qclose(9)	—
qbufcall(9)	—
qtimeout(9)	—
mi_bufcall(9)	—
putq(9)	—
putbq(9)	—
putnext(9)	—
qreply(9)	—

2. *Asynchronous*. These entry points, callbacks and callouts are *not* referenced against a *STREAMS* queue structure. That is, they were invoked without a specific *STREAMS* queue structure as an argument. These procedures are as follows:

bufcall(9)	—
esbbufcall(9)	—
timeout(9)	—
esballoc(9)	(free routine)

16.1 Configuration

SVR 4.2 MP specifies a synchronization mechanism that can be used during configuration of a *STREAMS* driver or module to specify the level of synchronization required by a module. The *SVR 4* synchronization levels are as follows:

SQLVL_DEFAULT	<i>Default level synchronization.</i> Specifies that the module uses the default synchronization scheme. This is the same as specifying SQLVL_MODULE.
---------------	---

SQLVL_GLOBAL	<i>Global (STREAMS scheduler) level synchronization.</i> Specifies that all of <i>STREAMS</i> can be access by only one thread at the same time. The module is run with global synchronization. This means that only one <i>STREAMS</i> executive thread will be permitted to enter any module. This makes the entire <i>STREAMS</i> executive single threaded and is useful primarily for debugging. This is the same as "Uniprocessor Emulation" on some systems, and reduces the <i>STREAMS</i> executive to running on a single processor at a time. This option should normally be used only for debugging.
SQLVL_ELSEWHERE	<i>Module group level synchronization.</i> Specifies that the module is run with synchronization within a group of modules. Only one thread of execution will be within the group of modules at a time. The group is separately specified as a character string name. This permits a group of modules to run single threaded as though they are running on a single processor, without interfering with the concurrency of other modules outside the group. This can be important for testing and for modules that implicitly share unprotected data structures.
SQLVL_MODULE	<i>Module level synchronization.</i> Specifies that all instances of a module can be accessed by only one thread at the same time. This is the default value. The module is run with synchronization at the module. Only one thread of execution will be permitted within the module. Where the module does not share data structures between modules, this has a similar effect on running on a uniprocessor system. This is the default and works best for non-multiprocessor-safe modules written in accordance with <i>STREAMS</i> guidelines. This level is roughly equivalent to <i>Solaris</i> D_MTPERMOD perimeters.
SQLVL_QUEUEPAIR	<i>Queue pair level synchronization.</i> Specifies that each queue pair can be accessed by only one thread at the same time. Only one thread will be permitted to enter a given queue's procedures within a given queue pair. Where the read and write side of the queue pair share the same private structure ('q->q_ptr'), this provides multiprocessor protection of the common data structure to all synchronous entry points without an external lock. This level is roughly equivalent to <i>Solaris</i> D_MTAPAIR perimeters.

SQLVL_QUEUE	<i>Queue level synchronization.</i> Specifies that each queue can be accessed by only one thread at the same time. The module is run with synchronization at the queue. Only one thread of execution will be permitted to enter a given queue's procedures, however, another thread will be permitted to enter procedures of the other queue in the queue pair. This is useful when the read and write side of a module are largely independent and do not require synchronization between sides of the queue pair. This level is roughly equivalent to <i>Solaris</i> D_MTPERQ perimeters.
SQLVL_NOP	<i>No synchronization.</i> Specifies that each queue can be accessed by more than one thread at a the same time. The protection of internal data and of <code>put(9)</code> and <code>srv(9)</code> procedures against <code>timeout(9)</code> or <code>bufcall(9)</code> is done by the module or driver itself. This synchronization level should be used essentially for multiprocessor-efficient modules. This level is roughly equivalent to <i>Solaris</i> D_MP flag.

16.2 Synchronous Entry Points

Synchronous Entry Points are those entry points into the *STREAMS* driver or module that will be synchronized according to the specified synchronization level.

put(9)	<i>Queue put procedure.</i> If the module has any synchronization level other than SQLVL_NOP, the put procedure will be exclusive. Attempts to enter the put procedure while another thread is running within the synchronization level will result in the call being postponed until the thread currently in the synchronization level exits.
srv(9)	If the module has any synchronization level other than SQLVL_NOP, <i>Queue service procedure.</i> the service procedure will be exclusive. Attempts to enter the service procedure while another thread is running within the synchronization level will result in the service procedure being postponed until the thread currently in the synchronization level exits.
qopen(9)	<i>Queue open procedure.</i> The queue open procedure is synchronous and exclusive before the call to <code>qprocson(9)</code> , or in any event, until return from the procedure. If the module has synchronization level of global, elsewhere or per-module; the call to the qopen procedure is exclusive.

<code>qclose(9)</code>	<i>Queue close procedure.</i> The queue close procedure is synchronous and exclusive after the call to <code>qprocsoff(9)</code> , or in any event, after return from the procedure. If the module has synchronization level of global, elsewhere or per-module; the call to the <code>qclose</code> procedure is exclusive.
<code>qprocson(9)</code>	<i>Queue procedures on.</i>
<code>qprocsoff(9)</code>	<i>Queue procedures off.</i>
<code>freezestr(9)</code>	<i>Freeze stream.</i>
<code>unfreezestr(9)</code>	<i>Thaw stream.</i>
<code>qwriter(9)</code>	<i>Queue writer.</i>

16.3 Synchronous Callbacks

Synchronous Callbacks are those callbacks into the *STREAMS* driver or module that will be synchronized according to the specified synchronization level. Synchronous callbacks are an extension to the *UNIX System V Release 4.2* specifications of *STREAMS*. Synchronous callback extensions include *Solaris* extensions and *AIX* extensions.

These include:

<code>qbufcall(9)</code>	– queue referenced buffer call
<code>qtimeout(9)</code>	– queue referenced timeout
<code>qunbufcall(9)</code>	– queue referenced buffer call cancel
<code>quntimeout(9)</code>	– queue referenced timeout cancel
<code>mi_bufcall(9)</code>	– queue reference buffer call

16.4 Synchronous Callouts

<code>putnext(9)</code>	–
<code>qreply(9)</code>	–

16.5 Asynchronous Entry Points

16.6 Asynchronous Callbacks

Asynchronous Callbacks are those callbacks into the *STREAMS* driver or module that will *not* be synchronized according to the specified synchronization level. Asynchronous callbacks are the basic *UNIX System V Release 4.2* callbacks.

16.7 Asynchronous Callouts

17 Administration

17.1 Administrative Utilities

<code>autopush(8)</code>	control the autopush module list for a <i>STREAMS</i> device
<code>fattach(8)</code>	name a <i>STREAMS</i> file
<code>fdetach(8)</code>	unlink a named <i>STREAMS</i> file
<code>insf(8)</code>	install special device files
<code>scls(8)</code>	produce a list of module and driver names
<code>strace(8)</code>	write <i>STREAMS</i> event trace messages to the standard output
<code>strclean(8)</code>	clean up after the <i>STREAMS</i> error and trace loggers
<code>strconf(8)</code>	<i>STREAMS</i> configuration utility
<code>streams_mknod(8)</code>	create or remove <i>STREAMS</i> device nodes
<code>strerr(8)</code>	receive error log messages from the <i>STREAMS</i> <code>log(4)</code> driver
<code>strinfo(8)</code>	display information about <i>STREAMS</i> devices
<code>strload(8)</code>	loads the <i>STREAMS</i> subsystem
<code>strsetup(8)</code>	<i>STREAMS</i> setup command
<code>strvf(8)</code>	<i>STREAMS</i> verification tool

17.1.1 autopush(8)

NAME

SYNOPSIS

DESCRIPTION

APPLICATION USAGE

FUTURE DIRECTIONS

SEE ALSO

COMPATIBILITY

CONFORMANCE

HISTORY

17.1.2 fattach(8)

NAME

SYNOPSIS

DESCRIPTION

APPLICATION USAGE

FUTURE DIRECTIONS

SEE ALSO

COMPATIBILITY

CONFORMANCE

HISTORY

17.1.3 fdetach(8)

NAME

SYNOPSIS

DESCRIPTION

APPLICATION USAGE

FUTURE DIRECTIONS

SEE ALSO

COMPATIBILITY

CONFORMANCE

HISTORY

17.1.4 insf(8)

NAME

SYNOPSIS

DESCRIPTION

APPLICATION USAGE

FUTURE DIRECTIONS

SEE ALSO

COMPATIBILITY

CONFORMANCE

HISTORY

17.1.5 scls(8)

NAME

SYNOPSIS

DESCRIPTION

APPLICATION USAGE

FUTURE DIRECTIONS

SEE ALSO

COMPATIBILITY

CONFORMANCE

HISTORY

17.1.6 strace(8)

NAME

SYNOPSIS

DESCRIPTION

APPLICATION USAGE

FUTURE DIRECTIONS

SEE ALSO

COMPATIBILITY

CONFORMANCE

HISTORY

17.1.7 strclean(8)

NAME

SYNOPSIS

DESCRIPTION

APPLICATION USAGE

FUTURE DIRECTIONS

SEE ALSO

COMPATIBILITY

CONFORMANCE

HISTORY

17.1.8 strconf(8)

NAME

SYNOPSIS

DESCRIPTION

APPLICATION USAGE

FUTURE DIRECTIONS

SEE ALSO

COMPATIBILITY

CONFORMANCE

HISTORY

17.1.9 streams_mknod(8)

NAME

SYNOPSIS

DESCRIPTION

APPLICATION USAGE

FUTURE DIRECTIONS

SEE ALSO

COMPATIBILITY

CONFORMANCE

HISTORY

17.1.10 strerr(8)

NAME

SYNOPSIS

DESCRIPTION

APPLICATION USAGE

FUTURE DIRECTIONS

SEE ALSO

COMPATIBILITY

CONFORMANCE

HISTORY

17.1.11 strinfo(8)

NAME

SYNOPSIS

DESCRIPTION

APPLICATION USAGE

FUTURE DIRECTIONS

SEE ALSO

COMPATIBILITY

CONFORMANCE

HISTORY

17.1.12 strload(8)

NAME

strload – loads the *STREAMS* subsystem

SYNOPSIS

```
strload [-u|-q|[-l]] [-f file] [-d list] [-m list]
strload {-h|--help}
strload {-V|--version}
strload {-C|--copying}
```

DESCRIPTION

Linux Fast-STREAMS drivers and modules are dynamically loaded and unloaded. To support this function, each driver and module must have a configuration routine that performs the necessary initialization and setup operations.

For compatibility with *AIX PSE*, *Linux Fast-STREAMS* provides the **strload** command to load *STREAMS* drivers and modules. After loading the driver or module, the **strload** command calls the driver or module entry point using the `SYS_CFGDD` or `SYS_CFGMOD` operations in `sysconfig(9)`.

Each *STREAMS* driver or module must eventually call the `str_install(9)` utility to link into *STREAMS*.

Commonly used drivers or modules can be placed in a configuration file, which controls the normal setup and tear-down of *Linux Fast-STREAMS*. The configuration file allows more flexibility when loading drivers or modules by providing user-specified nodes and arguments.

OPTIONS

- [-l]** Loads the referenced drivers and modules. (This is the default if the ‘-q’ and ‘-u’ flags are not specified.) If no configuration file, driver or modules are listed in the command options, only the *STREAMS* executive is loaded;
- u, --unload** Unloads the referenced drivers and modules. If no configuration file, drivers or modules are listed in the command options, the entire *STREAMS* executive is unloaded;
- q, --query** Queries the referenced drivers and modules. If no configuration file, drivers or modules are listed in the command options, only the *STREAMS* executive is queried;
- f, --file file** Specifies the file to use as the configuration file. If ‘-f’ is not specified, the default filename is ‘`/etc/strload.conf`’.
- d, --drivers list** Specifies a list of driver names to load or unload. *list* is a comma separated list of driver names.

-m, --modules *list*

Specifies a list of module names to load or unload. *list* is a comma separated list of module names.

Without any options, by default, **strload** loads the *STREAMS* executive and takes its configuration from the file `/etc/strload.conf`. Only one of `-l`, `-q` and `-u` are permitted.

FILE FORMAT

APPLICATION USAGE

FUTURE DIRECTIONS

SEE ALSO

COMPATIBILITY

CONFORMANCE

HISTORY

17.1.13 strsetup(8)

NAME

SYNOPSIS

DESCRIPTION

APPLICATION USAGE

FUTURE DIRECTIONS

SEE ALSO

COMPATIBILITY

CONFORMANCE

HISTORY

17.1.14 strvf(8)

NAME

SYNOPSIS

DESCRIPTION

APPLICATION USAGE

FUTURE DIRECTIONS

SEE ALSO

COMPATIBILITY

CONFORMANCE

HISTORY

17.2 System Controls

Linux Fast-STREAMS provides and supports a number of system controls that can be accessed using the `/proc/sys/streams/*` files, with the `sysctl(2)` interface, or using `sysctl(8)`.

The following *STREAMS* specific system controls are provided by *Linux Fast-STREAMS*:

sys.streams.cltime

Defines the default time interval (in milliseconds) that closing stream will linger for messages to clear its queues before finally closing. Valid values are zero (0) or greater (`MAXINT`). The default value is the traditional *UNIX* value of 15,000 milliseconds (15 seconds). This system control defines the default for all new streams. The value for a given stream can be changed with the `streamio(7)` IO control `I_CLTIME`, after the stream is opened and before the stream is closed.

sys.streams.max_apush

Defines the maximum number of modules that can be configured for autopush onto a *STREAMS* driver. Valid values are zero (0) or greater (`MAXINT`). The default value is the traditional *UNIX* value of 8 modules. This system control defines the maximum for all new autopush configurations. Existing autopush configurations are unchanged.

sys.streams.max_mblk

Defines the maximum number of combined message data blocks that will be held on the per-CPU free list between runs of `runqueues(9)`. Valid values are zero (0) or greater (`MAXINT`). The default value is the traditional *UNIX* value of 10 blocks. This system control defines the maximum for the next run of `runqueues(9)` for all CPUs. Increasing or decreasing this number may have a positive or negative performance impact.

sys.streams.max_stramod

Defines the maximum number of autopush structures that can be defined for the entire system. Valid values are zero (0) or greater (`MAXINT`). The default value is eight (8) ??????. This system control defines the system-wide maximum.

sys.streams.max_strdev

Defines the maximum number of streams devices. This is normally the maximum number of character devices (`MAX_CHRDEV`). Valid values are zero (0) or greater (`MAXINT`). The default value is `MAX_CHRDEV` (256). This is a read-only system control.

sys.streams.max_strmod

Defines the maximum number of streams modules. This is normally the maximum number of character devices (`MAX_CHRDEV`). Valid values are zero (0) or greater (`MAXINT`). The default value is `MAX_CHRDEV` (256). This is a read-only system control.

sys.streams.msg_priority

Defines whether allocation priority will be honoured or whether all allocations will be treated the same. See `allocb(9)`. When 'true' (non-zero), allocation

priority will be honoured when allocating message blocks. When ‘false’ (zero), allocation priority will be ignored. Valid values are zero (0) or non-zero. The default value is zero (0). Changing this value may have a positive or negative impact on system performance.

sys.streams.nband

Defines the number of queue bands available. Valid values are zero (0) or greater (MAXINT). The default value is the traditional *UNIX* value of 256. This system control defines the system wide value. This system control is read-only.

sys.streams.nstrmsgs

Defines the maximum number of combined message data blocks that will be allocated. Valid values are zero (0) or greater (MAXINT). The default value is 2^{12} . Changing this value may have a positive or negative impact on system performance. Setting this value to a small number may be useful for debugging *STREAMS* drivers and modules.

sys.streams.nstrpush

Defines the maximum number of modules that can be pushed on a stream. This includes both autopushed modules and modules pushed using the *I_PUSH* IO control of *streamio*(7). Valid values are zero (0) or greater (MAXINT). The default value is the traditional *UNIX* value of 64. This is the default values for all new streams. Existing streams are unaffected.

sys.streams.hiwat

Defines the default high water mark (in message bytes) for the stream head. Valid values are zero (0) or greater (MAXINT). The default value is 5120. This is the default value for all new stream heads. Existing stream heads are unaffected. The value on existing stream heads can be changed by a driver or module by sending an *M_SETOPTS* message to the stream head.

sys.streams.lowat

Defines the default low water mark (in message bytes) for the stream head. Valid values are zero (0) or greater (MAXINT). The default value is 1024. This is the default value for all new stream heads. Existing stream heads are unaffected. The value on existing stream heads can be changed by a driver or module by sending an *M_SETOPTS* message to the stream head.

sys.streams.maxpsz

Defines the maximum packet size accepted (in message bytes) for the stream head. Valid values are zero (0) or greater (MAXINT). The default value is 2^{12} . This is the default value for all new stream heads. Existing stream heads are unaffected. The value on existing stream heads can be changed by a driver or module by sending an *M_SETOPTS* message to the stream head.

sys.streams.minpsz

Defines the minimum packet size accepted (in message bytes) for the stream head. Valid values are zero (0) or greater (MAXINT). The default value is 0. This is the default value for all new stream heads. Existing stream heads are

unaffected. The value on existing stream heads can be changed by a driver or module by sending an `M_SETOPTS` message to the stream head.

sys.streams.reuse_fmodsw

Defines whether fmodsw table entries will be reused. When `'false'` (zero), fmodsw table entries will not be reused. When `'true'` (non-zero), fmodsw table entries will be reused. Valid values are zero (0) and non-zero. The default value is `'false'` (zero).

sys.streams.rtime

Defines the time interval (in milliseconds) that the stream head will wait to forward held messages when `SNDHOLD` is set for the stream head. Valid values are zero (0) or greater (`MAXINT`). The default value is 10 milliseconds (one clock tick). This is the system wide value for all streams. Changing this value may have a positive or negative impact on system performance.

sys.streams.strhold

Defines whether the `SNDHOLD` feature is active or not. When `'false'` (0), the `SNDHOLD` feature is deactivated. When `'true'` (non-zero), the `SNDHOLD` feature is activated. Valid values are zero (0) or non-zero. The default value is `'false'` (0). This is the default value for all new streams. Existing streams are unaffected. This setting can be examined and altered on an open stream using the `I_GWROPT` and `I_SWROPT` IO controls of `streamio(7)`.

sys.streams.strctlsz

Defines the maximum *STREAMS* control part size. Valid values are zero (0) or greater (`MAXINT`). The default value is 2^{12} . This is the system wide maximum. Existing allocations are unaffected by lowering this value. Changing this value can have an impact on users of `putpmsg(2)`.

sys.streams.strmsgsz

Defines the maximum *STREAMS* message size. Valid values are zero (0) or greater (`MAXINT`). The default value is 2^{18} . This is the system wide maximum. Existing allocations are unaffected by lowering this value. Changing this value can have an impact on users of `write(2)`, `writew(2)`, and `putpmsg(2)`.

sys.streams.strthresh

Defines the maximum amount of memory that will be allocated for use by the *STREAMS* subsystem via `kmem_alloc(9)` or `kmem_zalloc(9)`, or indirectly using `alloca(9)` or allocating other *STREAMS* data structures. Valid values are zero (0) or greater (`MAXINT`). This is the system wide maximum. Existing allocations are unaffected by lowering this value. Changing this value to a low value may have some use in debugging *STREAMS* drivers and modules.

17.3 /proc File System

Appendix A STREAMS Data Structures

This appendix summarizes data structures commonly encountered in *STREAMS* module and driver development. Most of the data structures given in this appendix are contained in ‘sys/stream.h’.

A.1 Stream Structures

A.1.1 streamtab

This structure defines a module or a driver.

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

A.2 Queue Structures

Two sets of queue structures form a module. The structures are `queue`, `qband`, `qinit`, `module_info`, and `module_stat` (optional).

A.2.1 queue

`queue` structure has the following format:

```
struct queue {
    struct qinit *q_qinfo;      /* procedures and limits for queue */
    struct msgb *q_first;       /* head of message queue for this
                                queue */
    struct msgb *q_last;        /* tail of message queue for this
                                queue */
    struct queue *q_next;       /* next queue in stream */
    struct queue *q_link;       /* to next queue for scheduling */
    _VOID *q_ptr;               /* to private data structure */
    ulong q_count;              /* number of bytes in queue */
    ulong q_flag;               /* queue state */
    long q_minpsz;              /* min packet size accepted by this
                                module */
    long q_maxpsz;              /* max packet size accepted by this
                                module */
    ulong q_hiwat;              /* queue high water mark for flow
                                control */
    ulong q_lowat;              /* queue low water mark for flow
                                control */
    struct qband *q_bandp;      /* separate flow information */
    unsigned char q_nband;       /* number of priority bands */
    unsigned char q_blocked;     /* number of bands flow controlled */
    unsigned char q_pad1[2];     /* reserved for future use */
    long q_pad2[2];             /* reserved for future use */
};

typedef struct queue queue_t;
```

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

- *q-qinfo*: *st_rdinit* and *st_wrinit* (or *st_muxrinit* and *st_muxwinit*) from *streamtab*
- *q_minpsz*, *q_maxpsz*, *q_hiwat*, *q_lowat* from *module_info*
- *q_ptr* optionally, by the driver/module open routine

A.2.2 qinit

qinit format is as follows:

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

A.2.3 module_info

module_info has the following:

```
struct module_info {
    ushort mi_idnum;             /* module ID number */
    char *mi_idname;             /* module name */
    long mi_minpsz;              /* min packet size accepted */
    long mi_maxpsz;              /* max packet size accepted */
    ulong mi_hiwat;              /* high water mark, for flow control */
    ulong mi_lowat;              /* low water mark, for flow control */
};
```

A.2.4 module_stat

The format of *module_stat* is:

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

Note that in the event these counts are calculated by modules or drivers, the counts will be cumulative over all instantiations of modules with the same *fmodsw* entry and drivers with the same *cdevsw* entry. (*cdevsw* and *fmodsw* tables are described in [Appendix E \[STREAMS Configuration\]](#), page 323.)

A.2.5 qband

The queue flow information for each band is contained in the following structure:

```

/* Structure that describes the separate information for each priority
 * band in the queue
 */

struct qband {
    struct qband *qb_next;      /* next band's info */
    ulong qb_count;            /* number of bytes in band */
    struct msgb *qb_first;      /* beginning of band's data */
    struct msgb *qb_last;      /* end of band's data */
    ulong qb_hiwat;            /* high water mark for band */
    ulong qb_lowat;            /* low water mark for band */
    ulong qb_flag;              /* flag, QB_FULL, denotes that a band
                                of data flow is flow controlled */
    long qb_pad1;              /* reserved for future use */
};

typedef struct qband qband_t;

/*
 * qband flags
 */
#define QB_FULL    0x01        /* band is considered full */
#define QB_WANTW   0x02        /* someone wants to write to band */
#define QB_BACK    0x04        /* queue has been back-enabled */

```

A.3 Message Structures

A message is composed of a linked list of triples, consisting of two structures (msgb and datab) and a data buffer.

A.3.1 msgb

```

/* the message block, msgb, structure */

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

    struct datab *b_datab;      /* data block */
    unsigned char b_band;        /* message priority */
    unsigned char b_pad1;
    unsigned short b_flag;
    long b_pad2;
};

typedef struct msgb mblk_t;
typedef struct datab dblk_t;
typedef struct free_rtn frtn_t;

/* Message flags. These are interpreted by the Stream head. */

#define MSGMARK    0x01        /* last byte of message is "marked" */
#define MSGNOLoop  0x02        /* don't loop message around to

```

Appendix A: STREAMS Data Structures

```
                                write-side of Stream */
#define MSGDELIM 0x04          /* message is delimited */
#define MSGNOGET 0x08          /* getq does not return message */
#define MSGATTEN 0x20          /* attention to on read side */
```

A.3.2 datab

```
/* data block, datab, structure */

struct datab {
    union {
        struct datab *freep;
        struct free_rtn *frtnp;
    } db_f; /* used internally */
    unsigned char *db_base; /* first byte of buffer */
    unsigned char *db_lim; /* last byte+1 of buffer */
    unsigned char db_ref; /* count of message pointing to this
                           block */
    unsigned char db_type; /* message type */
    unsigned char db_iswhat; /* status of message/data/buffer
                              triplet */
    unsigned int db_size; /* used internally */
    caddr_t db_msgaddr; /* triplet message header pointing to
                        datab */
    long db_filler; /* reserved for future use */
};

#define db_freep db_f.freep
#define db_freeep db_f.frtnp
```

A.4 Input Output Control Structures

A.4.1 iocblk

This is contained in an M_IOCTL message block:

```
struct iocblk {
    int ioc_cmd; /* ioctl command type */
    cred_t *ioc_cr; /* full credentials */
    uint ioc_id; /* ioctl id */
    uint ioc_count; /* count of bytes in data field */
    int ioc_error; /* error code */
    int ioc_rval; /* return value */
    long ioc_filler[4]; /* reserved for future use */
};

#define ioc_uid ioc_cr->cr_uid
#define ioc_gid ioc_cr->cr_gid
```

A.4.2 copyreq

This is used in M_COPYIN/M_COPYOUT messages:

```
struct copyreq {
    int cq_cmd; /* ioctl command (from ioc_cmd) */
    cred_t *cq_cr; /* full credentials */
    uint cq_id; /* ioctl id (from ioc_id) */
};
```



```

    caddr_t cq_addr;          /* address to copy data to/from */
    uint cq_size;             /* number of bytes to copy */
    int cq_flag;              /* see below */
    mblk_t *cq_private;       /* private state information */
    long cq_filler[4];        /* reserved for future use */
};

#define cq_uid cq_cr->cr_uid
#define cq_gid cq_cr->cr_gid

/* cq_flag values */

#define STRCANON    0x01      /* b_cont data block contains
                               canonical format specifier */
#define RECOPY     0x02      /* perform I_STR copyin again this
                               time using canonical format
                               specifier */

```

A.4.3 copyresp

This structure is used in M_IOCTLDATA:

```

struct copyresp {
    int cp_cmd;               /* ioctl command (from ioc_cmd) */
    cred_t *cp_cr;           /* full credentials */
    uint cp_id;              /* ioctl id (from ioc_id) */
    caddr_t cp_rval;         /* status of request: 0 for
                               successnon-zero for failure */
    uint cp_pad1;            /* reserved */
    int cp_pad2;             /* reserved */
    mblk_t *cp_private;      /* private state information */
    long cp_filler[4];       /* reserved for future use */
};

#define cp_uid cp_cr->cr_uid
#define cp_gid cp_cr->cr_gid

```

A.4.4 strioctl

This structure supplies user values as an argument to the ioctl call I_STR in streamio(7).

```

struct strioctl {
    int ic_cmd;               /* downstream request */
    int ic_timeout;          /* timeout acknowledgement - ACK/NAK */
    int ic_len;              /* length of data argument */
    char *ic_dp;             /* pointer to data argument */
};

```

A.5 Link Structures

A.5.1 linkblk

This structure is used in the lower multiplexor drivers to indicate a link.

```

/* this is used in lower multiplexor drivers to indicate a link */

struct linkblk {
    queue_t *l_qtop;         /* lowest level write queue of upper

```

```

                                Stream set to NULL for persistent
                                links */
    queue_t *l_qbot;           /* highest level write queue of lower
                                Stream */
    int l_index;               /* system-unique index for lower
                                Stream */
    long l_pad[5];            /* reserved for future use */
};

```

A.6 Options Structures

A.6.1 stroptions

This is an options structure of the `M_SETOPTS` message. The structure is sent upstream with modules or drivers to specify an optional stream head value.

```

struct stroptions {
    ulong so_flags;           /* options to set */
    short so_readopt;         /* read option */
    ushort so_wroff;         /* write offset */
    long so_minpsz;           /* minimum read packet size */
    long so_maxpsz;           /* maximum read packet size */
    ulong so_hiwat;           /* read queue high water mark */
    ulong so_lowat;           /* read queue low water mark */
    unsigned char so_band;    /* band for water marks */
};

/* flags for Stream options set message */
#define SO_ALL      0x003f    /* set all options */
#define SO_READOPT  0x0001    /* set read option */
#define SO_WROFF    0x0002    /* set write offset */
#define SO_MINPSZ   0x0004    /* set minimim packet size */
#define SO_MAXPSZ   0x0008    /* set maximim packet size */
#define SO_HIWAT    0x0010    /* set high water mark */
#define SO_LOWAT    0x0020    /* set low water mark */
#define SO_MREADON  0x0040    /* set read notification on */
#define SO_MREADOFF 0x0080    /* set read notification off */
#define SO_NDELON   0x0100    /* old TTY semantics for NDELAY
                                reads/writes */
#define SO_NDELOFF  0x0200    /* STREAMS semantics for NDELAY
                                reads/writes */
#define SO_ISTTY    0x0400    /* Stream is acting as terminal */
#define SO_ISNTTY   0x0800    /* Stream is not acting as a terminal
                                */
#define SO_TOSTOP   0x1000    /* stop on background writes to
                                Stream */
#define SO_TONSTOP  0x2000    /* don't stop on background jobs to
                                stream */
#define SO_BAND     0x4000    /* water marks that affect band */
#define SO_DELIM    0x8000    /* messages are delimited */
#define SO_NODELIM  0x010000  /* turn off delimiters */
#define SO_STRHOLD   0x020000  /* enable strwrite message coalescing
                                */

```

Appendix B STREAMS Message Types

B.1 Message Types

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

STREAMS does not prevent a module or driver from generating any message type and sending it in any direction on the *Stream*. However, established processing and direction rules should be observed. *Stream head* processing according to message type is fixed, although certain parameters can be altered.

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

Ordinary messages are also called normal or non-priority messages. Ordinary messages are subject to flow control whereas high priority messages are not.

B.2 Ordinary Messages

- | | |
|----------------|--|
| M_BREAK | Sent to a driver to request that <i>BREAK</i> be transmitted on whatever media the driver is controlling. The message format is not defined by <i>STREAMS</i> and its use is developer dependent. This message may be considered a special case of an M_CTL message. An M_BREAK message cannot be generated by a user-level process and is always discarded if passed to the <i>Stream head</i> . |
| M_CTL | Generated by modules that wish to send information to a particular module or type of module. M_CTL messages are typically used for inter-module communication, as when adjacent <i>STREAMS</i> protocol modules negotiate the terms of their interface. An M_CTL message cannot be generated by a user-level process and is always discarded if passed to the <i>Stream head</i> . |
| M_DATA | Intended to contain ordinary data. Messages allocated by the <code>alloca()</code> routine (see <i>STREAMS</i> Utilities) are type M_DATA by default. M_DATA messages are generally sent bidirectionally on a <i>Stream</i> and their contents can be passed between a process and the <i>Stream head</i> . In the <code>getmsg(2)</code> and <code>putmsg(2)</code> system calls, the contents of M_DATA message blocks are referred to as the data part. Messages composed of multiple message blocks will typically have M_DATA as the message type for all message blocks following the first. |
| M_DELAY | Sent to a media driver to request a real-time delay on output. The data buffer associated with this message is expected to contain an integer to indicate the number of machine ticks of delay desired. M_DELAY messages are typically used to prevent transmitted data from exceeding the buffering capacity of slower terminals. |

The message format is not defined by *STREAMS* and its use is developer dependent. Not all media drivers may understand this message. This message may be considered a special case of an *M_CTL* message. An *M_DELAY* message cannot be generated by a user-level process and is always discarded if passed to the *Stream head*.

M_IOCTL Generated by the *Stream head* in response to *I_STR*, *I_LINK*, *I_UNLINK*, *I_PLINK*, and *I_PUNLINK* [*ioctl(2)* *STREAMS* system calls, see *streamio(7)*], and in response to *ioctl* calls which contain a command argument value not defined in *streamio(7)*. When one of these *ioctls* is received from a user process, the *Stream head* uses values supplied in the call and values from the process to create an *M_IOCTL* message containing them, and sends the message downstream. *M_IOCTL* messages are intended to perform the general *ioctl* functions of character device drivers.

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

```
struct strioctl {
    int ic_cmd;           /* downstream request */
    int ic_timeout;       /* ACK/NAK timeout */
    int ic_len;           /* length of data arg */
    char *ic_dp;          /* ptr to data arg */
};
```

where *ic_cmd* is the request (or command) defined by a downstream module or driver, *ic_timeout* is the time the *Stream head* will wait for acknowledgement to the *M_IOCTL* message before timing out, and *ic_dp* is a pointer to an optional data buffer. On input, *ic_len* contains the length of the data in the buffer passed in and, on return from the call, it contains the length of the data, if any, being returned to the user in the same buffer.

The *M_IOCTL* message format is one *M_IOCTL* message block followed by zero or more *M_DATA* message blocks. *STREAMS* constructs an *M_IOCTL* message block by placing an *iocblk* structure, defined in ‘*sys/stream.h*’, in its data buffer (see [Appendix A \[STREAMS Data Structures\]](#), page 275, for a complete *iocblk* structure):

```
struct iocblk {
    int ioc_cmd;           /* ioctl command type */
    cred_t *ioc_cr;        /* full credentials */
    uint ioc_id;           /* ioctl identifier */
    uint ioc_count;        /* byte count for ioctl data */
    int ioc_error;         /* error code for M_IOCACK or
                           M_IOCNAK */
    int ioc_rval;          /* return value for M_IOCACK */
    long ioc_filler[4];    /* reserved for future use */
};
```

For an *I_STR* *ioctl*, *ioc_cmd* corresponds to *ic_cmd* of the *strioctl* structure. *ioc_cr* points to a credentials structure defining the user process’s permissions (see ‘*cred.h*’). Its contents can be tested to determine if the user issuing the *ioctl* call is authorized to do so. For an *I_STR* *ioctl*, *ioc_count* is the number of data bytes, if any, contained in the message and corresponds to *ic_len*.

ioc_id is an identifier generated internally, and is used by the *Stream head* to match each `M_IOCTL` message sent downstream with response messages sent upstream to the *Stream head*. The response message which completes the *Stream head* processing for the `ioctl` is an `M_IOCACK` (positive acknowledgement) or an `M_IOCNAK` (negative acknowledgement) message.

For an `I_STR ioctl`, if a user supplies data to be sent downstream, the *Stream head* copies the data, pointed to by *ic_dp* in the `striocctl` structure, into `M_DATA` message blocks and links the blocks to the initial `M_IOCTL` message block. *ioc_count* is copied from *ic_len*. If there are no data, *ioc_count* is zero.

If the *Stream head* does not recognize the command argument of an `ioctl`, it creates a transparent `M_IOCTL` message. The format of a transparent `M_IOCTL` message is one `M_IOCTL` message block followed by one `M_DATA` block. The form of the `iocblk` structure is the same as above. However, *ioc_cmd* is set to the value of the command argument in the `ioctl` system call and *ioc_count* is set to 'TRANSPARENT', defined in 'sys/stream.h'. 'TRANSPARENT' distinguishes the case where an `I_STR ioctl` may specify a value of *ioc_cmd* equivalent to the command argument of a transparent `ioctl`. The `M_DATA` block of the message contains the value of the *arg* parameter in the `ioctl` call.

The first module or driver that understands the *ioc_cmd* request contained in the `M_IOCTL` acts on it. For an `I_STR ioctl`, this action generally includes an immediate upstream transmission of an `M_IOCACK` message. For transparent `M_IOCTLs`, this action generally includes the upstream transmission of an `M_COPYIN` or `M_COPYOUT` message.

Intermediate modules that do not recognize a particular request must pass the message on. If a driver does not recognize the request, or the receiving module can not acknowledge it, an `M_IOCNAK` message must be returned.

`M_IOCACK` and `M_IOCNAK` message types have the same format as an `M_IOCTL` message and contain an `iocblk` structure in the first block. An `M_IOCACK` block may be linked to following `M_DATA` blocks. If one of these messages reaches the *Stream head* with an identifier which does not match that of the currently-outstanding `M_IOCTL` message, the response message is discarded. A common means of assuring that the correct identifier is returned is for the replying module to convert the `M_IOCTL` message into the appropriate response type and set *ioc_count* to 0, if no data are returned. Then, the `qreply(9)` utility (see [Appendix C \[STREAMS Utilities\], page 295](#)) is used to send the response to the *Stream head*.

In an `M_IOCACK` or `M_IOCNAK` message, *ioc_error* holds any return error condition set by a downstream module. If this value is non-zero, it is returned to the user in `errno(3)`. Note that both an `M_IOCNAK` and an `M_IOCACK` may return an error.¹ However, only an `M_IOCACK` can have a return value. For an `M_IOCACK`, *ioc_rval* holds any return value set by a responding module. For an `M_IOCNAK`,

¹ Linux Fast-STREAMS implementation does not permit `M_IOCACK` to return an error number and a return value simultaneously.

ioc_rval is ignored by the *Stream head*. If a module processing an *I_STR ioctl* wants to send data to a user process, it must use the *M_IOCACK* message which it constructs such that the *M_IOCACK* block is linked to one or more following *M_DATA* blocks containing the user data. The module must set *ioc_count* to the number of data bytes sent. The *Stream head* places the data in the address pointed to by *ic_dp* in the user *I_STR strioctl* structure.

If a module processing a transparent *ioctl* (i.e., it received a transparent *M_IOCTL*) wants to send data to a user process, it can use only an *M_COPYOUT* message. For a transparent *ioctl*, no data can be sent to the user process in an *M_IOCACK* message. All data must have been sent in a preceding *M_COPYOUT* message. The *Stream head* will ignore any data contained in an *M_IOCACK* message (in *M_DATA* blocks) and will free the blocks. No data can be sent with an *M_IOCNAK* message for any type of *M_IOCTL*. The *Stream head* will ignore and will free any *M_DATA* blocks.

The *Stream head* blocks the user process until an *M_IOCACK* or *M_IOCNAK* response to the *M_IOCTL* (same *ioc_id*) is received. For an *M_IOCTL* generated from an *I_STR ioctl*, the *Stream head* will "time out" if no response is received in *ic_timeout* interval (the user may specify an explicit interval or specify use of the default interval). For *M_IOCTL* messages generated from all other *ioctls*, the default (infinite) is used.²

M_PASSFP Used by *STREAMS* to pass a file pointer from the *Stream head* at one end of a *Stream* pipe to the *Stream head* at the other end of the same *Stream* pipe.

The message is generated as a result of an *I_SENDFD ioctl* [see *streamio(7)*] issued by a process to the sending *Stream head*. *STREAMS* places the *M_PASSFP* message directly on the destination *Stream head*'s read queue to be retrieved by an *I_RECVFD ioctl* [see *streamio(7)*]. The message is placed without passing it through the *Stream* (i.e., it is not seen by any modules or drivers in the *Stream*). This message should never be present on any queue except the read queue of a *Stream head*. Consequently, modules and drivers do not need to recognize this message, and it can be ignored by module and driver developers.

M_PROTO Intended to contain control information and associated data. The message format is one or more (see note) *M_PROTO* message blocks followed by zero or more *M_DATA* message blocks as shown in [Figure B.1](#). The semantics of the *M_DATA* and *M_PROTO* message block are determined by the *STREAMS* module that receives the message.

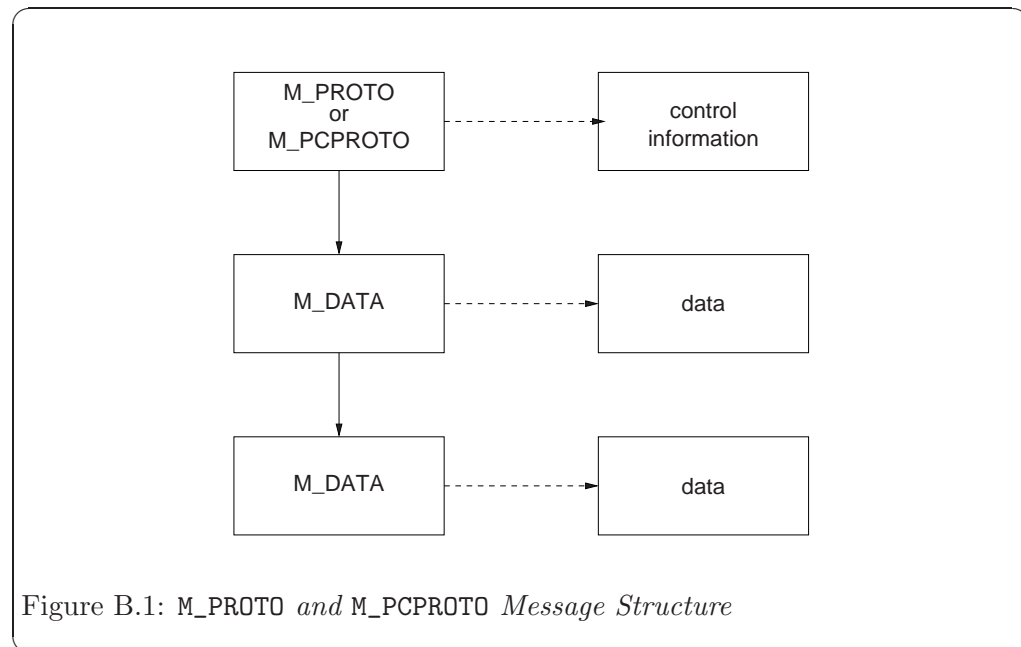
The *M_PROTO* message block will typically contain implementation dependent control information. *M_PROTO* messages are generally sent bidirectionally on a *Stream*, and their contents can be passed between a process and the *Stream head*. The contents of the first message block of an *M_PROTO* message is generally referred to as the control part, and the contents of any following *M_DATA* message

² Linux Fast-STREAMS uses a default of '15' seconds, in accordance with common practice.

blocks are referred to as the data part. In the `getmsg(2)` and `putmsg(2)` system calls, the control and data parts are passed separately.

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

Although its use is not recommended, the format of `M_PROTO` and `M_PCPROTO` (generically *PROTO*) messages sent upstream to the *Stream head* allows multiple *PROTO* blocks at the beginning of the message. `getmsg(2)` will compact the blocks into a single control part when passing them to the user process.



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

M_SETOPTS

Used to alter some characteristics of the *Stream head*. It is generated by any downstream module, and is interpreted by the *Stream head*. The data buffer of the message has the following structure (see [Appendix A \[STREAMS Data Structures\]](#), page 275, for a complete `stroptions` structure):

```

struct stroptions {
    ulong so_flags;           /* options to set */
    short so_readopt;         /* read option */
    ushort so_wroff;         /* write offset */
    long so_minpsz;           /* minimum read packet size */
    long so_maxpsz;           /* maximum read packet size */
    ulong so_hiwat;           /* read queue high-water mark */
    ulong so_lowat;           /* read queue low-water mark */
    unsigned char so_band;    /* update water marks for this band */
};
  
```


where *so_flags* specifies which options are to be altered, and can be any combination of the following:

SO_ALL Update all options according to the values specified in the remaining fields of the **stroptions** structure.

SO_READOPT

Set the read mode [see **read(2)**] to **RNORM** (byte stream), **RMSGD** (message discard), **RMSGN** (message non-discard), **RPROTNORM** (normal protocol), **RPROTDAT** (turn **M_PROTO** and **M_PCPROTO** messages into **M_DATA** messages), or **RPROTDIS** (discard **M_PROTO** and **M_PCPROTO** blocks in a message and retain any linked **M_DATA** blocks) as specified by the value of *so_readopt*.

SO_WROFF Direct the *Stream head* to insert an offset (unwritten area, see "write Offset" in see [\[**undefined**\] \[**undefined**\]](#), page [\[**undefined**\]](#)) specified by *so_wroff* into the first message block of all **M_DATA** messages created as a result of a **write(2)** system call. The same offset is inserted into the first **M_DATA** message block, if any, of all messages created by a **putmsg(2)** system call. The default offset is zero. The offset must be less than the maximum message buffer size (system dependent). Under certain circumstances, a write offset may not be inserted.

A module or driver must test that *b_rptr* in the **msgb** structure is greater than *db_base* in the **atab** structure to determine that an offset has been inserted in the first message block.

SO_MINPSZ

Change the minimum packet size value associated with the *Stream head* read queue to *so_minpsz* (see *q_minpsz* in the **queue** structure, see [Appendix A \[STREAMS Data Structures\]](#), page 275). This value is advisory for the module immediately below the *Stream head*. It is intended to limit the size of **M_DATA** messages that the module should put to the *Stream head*. There is no intended minimum size for other message types. The default value in the *Stream head* is zero.

SO_MAXPSZ

Change the maximum packet size value associated with the *Stream head* read queue to *so_maxpsz* (see *q_maxpsz* in the **queue** structure, see [Appendix A \[STREAMS Data Structures\]](#), page 275). This value is advisory for the module immediately below the *Stream head*. It is intended to limit the size of **M_DATA** messages that the module should put to the *Stream head*. There is no intended maximum size for other message types. The default value in the *Stream head* is 'INFPsz', the maximum *STREAMS* allows.

- SO_HIWAT** Change the flow control high water mark (*q.hiwat* in the **queue** structure, *qb.hiwat* in the **qband** structure) on the *Stream head* read queue to the value specified in *so.hiwat*.
- SO_LOWAT** Change the flow control low water mark (*q.lowat* in the **queue** structure, *qb.lowat* in the **qband** structure) on the *Stream head* read queue to the value specified in *so.lowat*.
- SO_MREADON**
Enable the *Stream head* to generate **M_READ** messages when processing a **read(2)** system call. If both **SO_MREADON** and **SO_MREADOFF** are set in *so.flags*, **SO_MREADOFF** will have precedence.
- SO_MREADOFF**
Disable the *Stream head* generation of **M_READ** messages when processing a **read(2)** system call. This is the default. If both **SO_MREADON** and **SO_MREADOFF** are set in *so.flags*, **SO_MREADOFF** will have precedence.
- SO_NDELOFF**
Set non-*STREAMS* *tty* semantics for **O_NDELAY** (or **O_NONBLOCK**) processing on **read(2)** and **write(2)** system calls. If **O_NDELAY** (or **O_NONBLOCK**) is set, a **read(2)** will return '0' if no data are waiting to be read at the *Stream head*. If **O_NDELAY** (or **O_NONBLOCK**) is clear, a **read(2)** will block until data become available at the *Stream head*. (See note below.) Regardless of the state of **O_NDELAY** (or **O_NONBLOCK**), a **write(2)** will block on flow control and will block if buffers are not available.
If both **SO_NDELOFF** and **SO_NDELOFF** are set in *so.flags*, **SO_NDELOFF** will have precedence.
NOTE: For conformance with the *POSIX* standard, it is recommended that new applications use the **O_NONBLOCK** flag whose behavior is the same as that of **O_NDELAY** unless otherwise noted.
- SO_NDELOFF**
Set *STREAMS* semantics for **O_NDELAY** (or **O_NONBLOCK**) processing on **read(2)** and **write(2)** system calls. If **O_NDELAY** (or **O_NONBLOCK**) is set, a **read(2)** will return -1 and set **[EAGAIN]** in *errno* if no data are waiting to be read at the *Stream head*. If **O_NDELAY** (or **O_NONBLOCK**) is clear, a **read(2)** will block until data become available at the *Stream head*. (See note above)
If **O_NDELAY** (or **O_NONBLOCK**) is set, a **write(2)** will return -1 and set **[EAGAIN]** in *errno* if flow control is in effect when the call is received. It will block if buffers are not available. If **O_NDELAY** (or **O_NONBLOCK**) is set and part of the buffer has been written and a flow control or buffers not available condition is encountered, **write(2)** will terminate and return the number of bytes written.

If `O_NDELAY` (or `O_NONBLOCK`) is clear, a `write(2)` will block on flow control and will block if buffers are not available. This is the default. If both `SO_NDELO` and `SO_NDELOFF` are set in `so_flags`, `SO_NDELOFF` will have precedence.

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

SO_BAND Set water marks in a band. If the `SO_BAND` flag is set with the `SO_HIWAT` or `SO_LOWAT` flag, the `so_band` field contains the priority band number the `so_hiwat` and `so_lowat` fields pertain to.

If the `SO_BAND` flag is not set and the `SO_HIWAT` and `SO_LOWAT` flags are on, the normal high and low water marks are affected. The `SO_BAND` flag has no effect if `SO_HIWAT` and `SO_LOWAT` flags are off. Only one band's water marks can be updated with a single `M_SETOPTS` message.

SO_ISTTY Inform the *Stream head* that the *Stream* is acting like a controlling terminal.

For `SO_ISTTY`, the *Stream* may or may not be allocated as a controlling terminal via an `M_SETOPTS` message arriving upstream during open processing. If the *Stream head* is opened before receiving this message, the *Stream* will not be allocated as a controlling terminal until it is queued again by a session leader.

SO_ISNTTY

Inform the *Stream head* that the *Stream* is no longer acting like a controlling terminal.

SO_TOSTOP

Stop on background writes to the *Stream*.

SO_TONSTOP

Do not stop on background writes to the *Stream*. `SO_TOSTOP` and `SO_TONSTOP` are used in conjunction with job control.

M_SIG

Sent upstream by modules or drivers to post a signal to a process. When the message reaches the front of the *Stream head* read queue, it evaluates the first data byte of the message as a signal number, defined in `<sys/signal.h>`. (Note that the signal is not generated until it reaches the front of the *Stream head* read queue.) The associated signal will be sent to process(es) under the following conditions:

If the signal is `{SIGPOLL}`, it will be sent only to those processes that have explicitly registered to receive the signal [see `I_SETSIG` in `streamio(7)`].

If the signal is not `{SIGPOLL}` and the *Stream* containing the sending module or driver is a controlling *tty*, the signal is sent to the associated process group. A *Stream* becomes the controlling *tty* for its process group if, on `open(2)`, a

module or driver sends an `M_SETOPTS` message to the *Stream head* with the `SO_ISTTY` flag set.

If the signal is not `{SIGPOLL}` and the *Stream* is not a controlling *tty*, no signal is sent, except in case of `SIOCSPGRP` and `TIOCSPGRP`. These two `ioctl`s set the process group field in the *Stream head* so the *Stream* can generate signals even if it is not a controlling *tty*.

B.3 High Priority Messages

M_COPYIN Generated by a module or driver and sent upstream to request that the *Stream head* perform a `copyin()` on behalf of the module or driver. It is valid only after receiving an `M_IOCTL` message and before an `M_IOCACK` or `M_IOCNAK`.

The message format is one `M_COPYIN` message block containing a `copyreq` structure, defined in `<sys/stream.h>` (see *STREAMS Data Structures* for a complete `copyreq` structure):

```
struct copyreq {
    int cq_cmd;                /* ioctl command (from ioc_cmd) */
    cred_t *cq_cr;            /* full credentials */
    unit cq_id;               /* ioctl id (from ioc_id) */
    caddr_t cq_addr;          /* address to copy data to/from */
    unit cq_size;             /* number of bytes to copy */
    int cq_flag mblk_t *cq_private; /* private state information */
    long cq_filler[4];        /* reserved for future use */
};
```

The first three members of the structure correspond to those of the `iocblk` structure in the `M_IOCTL` message which allows the same message block to be reused for both structures. The *Stream head* will guarantee that the message block allocated for the `M_IOCTL` message is large enough to contain a `copyreq` structure. The `cq-addr` field contains the user space address from which the data are to be copied. The `cq-size` field is the number of bytes to copy from user space.

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

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

M_COPYOUT

Generated by a module or driver and sent upstream to request that the *Stream head* perform a `copyout()` on behalf of the module or driver. It is valid only after receiving an `M_IOCTL` message and before an `M_IOCACK` or `M_IOCNAK`.

The message format is one `M_COPYOUT` message block followed by one or more `M_DATA` blocks. The `M_COPYOUT` message block contains a `copyreq` structure as

described in the `M_COPYIN` message with the following differences: The `cq_addr` field contains the user space address to which the data are to be copied. The `cq_size` field is the number of bytes to copy to user space.

Data to be copied to user space is contained in the linked `M_DATA` blocks.

This message should not be queued by a module or driver unless it intends to process the data for the `ioctl` in some way.

M_ERROR Sent upstream by modules or drivers to report some downstream error condition. When the message reaches the *Stream head*, the *Stream* is marked so that all subsequent system calls issued to the *Stream*, excluding `close(2)` and `poll(2)`, will fail with `errno` set to the first data byte of the message. `POLLERR` is set if the *Stream* is being polled [see `poll(2)`]. All processes sleeping on a system call to the *Stream* are awakened. An `M_FLUSH` message with `FLUSHRW` is sent downstream.

The *Stream head* maintains two error fields, one for the read-side and one for the write-side. The one-byte format `M_ERROR` message sets both of these fields to the error specified by the first byte in the message.

The second style of the `M_ERROR` message is two bytes long. The first byte is the read error and the second byte is the write error. This allows modules to set a different error on the read-side and write-side. If one of the bytes is set to `NOERROR`, then the field for the corresponding side of the *Stream* is unchanged. This allows a module to just an error on one side of the *Stream*. For example, if the *Stream head* was not in an error state and a module sent an `M_ERROR` message upstream with the first byte set to `[EPROTO]` and the second byte set to `NOERROR`, all subsequent read-like system calls (for example, `read`, `getmsg`) will fail with `[EPROTO]`, but all write-like system calls (for example, `write`, `putmsg`) will still succeed. If a byte is set to 0, the error state is cleared for the corresponding side of the *Stream*. The values `NOERROR` and 0 are not valid for the one-byte form of the `M_ERROR` message.

M_FLUSH Requests all modules and drivers that receive it to flush their message queues (discard all messages in those queues) as indicated in the message. An `M_FLUSH` can originate at the *Stream head*, or in any module or driver. The first byte of the message contains flags that specify one of the following actions:

<code>FLUSHR:</code>	Flush the read queue of the module.
<code>FLUSHW:</code>	Flush the write queue of the module.
<code>FLUSHRW:</code>	Flush both the read queue and the write queue of the module.
<code>FLUSHBAND:</code>	Flush the message according to the priority associated with the band.

Each module passes this message to its neighbor after flushing its appropriate queue(s), until the message reaches one of the ends of the *Stream*.

Drivers are expected to include the following processing for `M_FLUSH` messages. When an `M_FLUSH` message is sent downstream through the write queues in a *Stream*, the driver at the *Stream* end discards it if the message action indicates that the read queues in the *Stream* are not to be flushed (only `FLUSHW` set). If

the message indicates that the read queues are to be flushed, the driver shuts off the `FLUSHW` flag, and sends the message up the *Stream*'s read queues. When a flush message is sent up a *Stream*'s read-side, the *Stream head* checks to see if the write-side of the *Stream* is to be flushed. If only `FLUSHR` is set, the *Stream head* discards the message. However, if the write-side of the *Stream* is to be flushed, the *Stream head* sets the `M_FLUSH` flag to `FLUSHW` and sends the message down the *Stream*'s write side. All modules that enqueue messages must identify and process this message type.

If `FLUSHBAND` is set, the second byte of the message contains the value of the priority band to flush.

M_HANGUP Sent upstream by a driver to report that it can no longer send data upstream. As example, this might be due to an error, or to a remote line connection being dropped. When the message reaches the *Stream head*, the *Stream* is marked so that all subsequent `write(2)` and `putmsg(2)` system calls issued to the *Stream* will fail and return an `[ENXIO]` error. Those `ioctl`s that cause messages to be sent downstream are also failed. `POLLHUP` is set if the *Stream* is being polled [see `poll(2)`].

However, subsequent `read(2)` or `getmsg(2)` calls to the *Stream* will not generate an error. These calls will return any messages (according to their function) that were on, or in transit to, the *Stream head* read queue before the `M_HANGUP` message was received. When all such messages have been read, `read(2)` will return 0 and `getmsg(2)` will set each of its two length fields to 0.

This message also causes a `{SIGHUP}` signal to be sent to the controlling process instead of the foreground process group, since the allocation and deallocation of controlling terminals to a session is the responsibility of the controlling process.

M_IOCACK Signals the positive acknowledgement of a previous `M_IOCTL` message. The message format is one `M_IOCACK` block (containing an `iocblk` structure, see `M_IOCTL`) followed by zero or more `M_DATA` blocks. The `iocblk` data structure may contain a value in `ioc_rval` to be returned to the user process. It may also contain a value in `ioc_error` to be returned to the user process in `errno`.

If this message is responding to an `I_STR` `ioctl` [see `streamio(7)`], it may contain data from the receiving module or driver to be sent to the user process. In this case, message format is one `M_IOCACK` block followed by one or more `M_DATA` blocks containing the user data. The *Stream head* returns the data to the user if there is a corresponding outstanding `M_IOCTL` request. Otherwise, the `M_IOCACK` message is ignored and all blocks in the message are freed.

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

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

M_IOCADATA

Generated by the *Stream head* and sent downstream as a response to an M_COPYIN or M_COPYOUT message. The message format is one M_IOCADATA message block followed by zero or more M_DATA blocks. The M_IOCADATA message block contains a copyresp structure, defined in <sys/stream.h> (see *STREAMS Data Structures* for a complete copyresp structure):

```

struct copyresp {
    int cp_cmd;           /* ioctl command (from ioc_cmd) */
    cred_t *cp_cr;        /* full credentials */
    unit cp_id;           /* ioctl id (from ioc_id) */
    caddr_t cp_rval;      /* status of request: 0 -> success
                          non_zero -> failure */
    unit cp_pad1;         /* reserved */
    int cp_pad2;          /* reserved */
    mblk_t *cp_private;   /* private state info (from
                          cq_private) */
    long cp_filler[4];    /* reserved for future use */
};

```

The first three members of the structure correspond to those of the iocblk structure in the M_IOCTL message which allows the same message blocks to be reused for all of the related transparent messages (M_COPYIN, M_COPYOUT, M_IOCACK, M_IOCNAK). The cp_rval field contains the result of the request at the *Stream head*. Zero indicates success and non-zero indicates failure. If failure is indicated, the module should not generate an M_IOCNAK message. It must abort all ioctl processing, clean up its data structures, and return.

The cp_private field is copied from the cq_private field in the associated M_COPYIN or M_COPYOUT message. It is included in the M_IOCADATA message so the message can be self-describing. This is intended to simplify ioctl processing by modules and drivers.

If the message is in response to an M_COPYIN message and success is indicated, the M_IOCADATA block will be followed by M_DATA blocks containing the data copied in. If an M_IOCADATA block is reused, any unused fields defined for the resultant message block should be cleared (particularly in an M_IOCACK or M_IOCNAK).

This message should not be queued by a module or driver unless it intends to process the data for the ioctl in some way.

M_IOCNAK Signals the negative acknowledgement (failure) of a previous M_IOCTL message. Its form is one M_IOCNAK block containing an iocblk data structure (see M_IOCTL). The iocblk structure may contain a value in ioc_error to be returned to the user process in errno. Unlike the M_IOCACK, no user data or return value can be sent with this message. If any M_DATA blocks follow the M_IOCNAK block, the *Stream head* will ignore and free them. When the *Stream head* receives an M_IOCNAK, the outstanding ioctl request, if any, will fail. The format and usage of this message type is described further under M_IOCTL.

M_PCPROTO

As the **M_PROTO** message type, except for the priority and the following additional attributes.

When an **M_PCPROTO** message is placed on a queue, its service procedure is always enabled. The *Stream head* will allow only one **M_PCPROTO** message to be placed in its read queue at a time. If an **M_PCPROTO** message is already in the queue when another arrives, the second message is silently discarded and its message blocks freed.

This message is intended to allow data and control information to be sent outside the normal flow control constraints.

The `getmsg(2)` and `putmsg(2)` system calls refer to **M_PCPROTO** messages as high priority messages.

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

M_PCSIG As the **M_SIG** message, except for the priority.

M_PCSIG is often preferable to the **M_SIG** message especially in *tty* applications, because **M_SIG** may be queued while **M_PCSIG** is more guaranteed to get through quickly. For example, if one generates an **M_SIG** message when the DEL (delete) key is hit on the terminal and one has already typed ahead, the **M_SIG** message becomes queued and the user doesn't get the call until it's too late; it becomes impossible to kill or interrupt a process by hitting a delete key.

M_READ Generated by the *Stream head* and sent downstream for a `read(2)` system call if no messages are waiting to be read at the *Stream head* and if read notification has been enabled. Read notification is enabled with the **SO_MREADON** flag of the **M_SETOPTS** message and disabled by use of the **SO_MREADOFF** flag. The message content is set to the value of the `nbyte` parameter (the number of bytes to be read) in the `read(2)` call.

M_READ is intended to notify modules and drivers of the occurrence of a read. It is also intended to support communication between Streams that reside in separate processors. The use of the **M_READ** message is developer dependent. Modules may take specific action and pass on or free the **M_READ** message. Modules that do not recognize this message must pass it on. All other drivers may or may not take action and then free the message.

This message cannot be generated by a user-level process and should not be generated by a module or driver. It is always discarded if passed to the *Stream head*.

M_START

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

The message format is not defined by *STREAMS* and its use is developer dependent. These messages may be considered special cases of an **M_CTL** message.

Appendix B: STREAMS Message Types

These messages cannot be generated by a user-level process and each is always discarded if passed to the *Stream head*.

M_STARTI

M_STOPI As M_START and M_STOP except that M_STARTI and M_STOPI are used to start and stop input.

Appendix C STREAMS Utilities

This appendix specifies the set of utility routines provided by STREAMS to assist development of modules and drivers.

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

The utility routines contained in this appendix represent an interface that will be maintained in subsequent versions of UNIX[®] System V. Other than these utilities (also see the section titled " Accessible Symbols and Functions " in Overview of Modules and Drivers), functions contained in the STREAMS kernel code may change between versions. UNIX[®] System V. Other than these utilities (also see the section titled " Accessible Symbols and Functions " in Overview of Modules and Drivers), functions contained in the STREAMS kernel code may change between versions.

Structure definitions are contained in STREAMS Data Structures . Routine references are found in this appendix. The following definitions are used:

- Blocked* A queue that can not be enabled due to flow control.
- Enable* To schedule a queue's service procedure to run.
- Free* To deallocate a STREAMS message or other data structure.
- Message block (bp)*
 A triplet consisting of an msgb structure, a datab structure, and a data buffer.
 It is referenced by its type definition mblk_t.
- Message (mp)*
 One or more linked message blocks. A message is referenced by its first message block.
- Message queue*
 Zero or more linked messages associated with a queue (queue structure).
- Queue (q)* A queue structure. When it appears with "message" in certain utility description lines, it means "message queue."
- Schedule* To place a queue on the internal linked list of queues which will subsequently have their service procedure called by the STREAMS scheduler.

The word module will generally mean module and/or driver. The phrase "next/following module" will generally refer to a module, driver, or Stream head.

C.1 Utility Descriptions

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

C.1.1 adjmsg-trim bytes in a message

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

`adjmsg(9)` trims bytes from either the head or tail of the message specified by *mp*. If *len* is greater than zero, it removes '*len*' bytes from the beginning of *mp*. If *len* is less than zero, it removes '*(-)**len*' bytes from the end of *mp*. If *len* is zero, `adjmsg(9)` does nothing.

`adjmsg(9)` only trims bytes across message blocks of the same type. It fails if *mp* points to a message containing fewer than *len* bytes of similar type at the message position indicated.

`adjmsg(9)` returns '1' on success and '0' on failure.

C.1.2 allocb-allocate a message and data block

```
struct msgb *
allocb(size, pri)
    int size;
    uint pri;
```

`allocb(9)` returns a pointer to a message block of type `M_DATA`, in which the data buffer contains at least *size* bytes. *pri* is one of `BPRI_LO`, `BPRI_MED`, or `BPRI_HI` and indicates how critically the module needs the buffer. *pri* is currently unused and is maintained only for compatibility with applications developed prior to *UNIX[®] System V Release 4.0*. If a block can not be allocated as requested, `allocb(9)` returns a 'NULL' pointer. *UNIX[®] System V Release 4.0*. If a block can not be allocated as requested, `allocb(9)` returns a 'NULL' pointer.

When a message is allocated via `allocb(9)` the *b_band* field of the `mbblk_t` is initially set to zero. Modules and drivers may set this field if so desired.

C.1.3 backq-get pointer to the queue behind a given queue

```
queue_t *
backq(q)
    register queue_t *q;
```

`backq(9)` returns a pointer to the queue behind a given queue. That is, it returns a pointer to the queue whose *q_next* (see [Section A.2.1 \[queue\]](#), page 275) pointer is *q*. If no such queue exists (as when *q* is at a *Stream end*), `backq(9)` returns 'NULL'.

C.1.4 bcanput-test for flow control in the given priority

```
int
bcanput(q, pri)
    register queue_t *q;
    unsigned char pri;
```

`bcanput(9)` provides modules and drivers with a way to test flow control in the given priority band. It returns '1' if a message of priority *pri* can be placed on the queue. It returns '0' if the priority band is flow controlled and sets the `QWANTW` flag to zero band (`QB_WANTW` to nonzero band).

If the band does not yet exist on the queue in question, '1' is returned.

The call '`bcanput(q, 0)`' is equivalent to the call '`canput(q)`'.

C.1.5 bufcall-recover from failure of allocb

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

bufcall(9) is provided to assist in the event of a block allocation failure. If **allocb(9)** returns ‘NULL’, indicating a message block is not currently available, **bufcall(9)** may be invoked.

bufcall(9) arranges for ‘(*func)(arg)’ to be called when a buffer of *size* bytes is available. *pri* is as described in **allocb(9)**. When *func* is called, it has no user context. It cannot reference the **current task_struct** structure, and must return without sleeping. **bufcall(9)** does not guarantee that the desired buffer will be available when *func* is called since interrupt processing may acquire it.

bufcall(9) returns ‘1’ on success, indicating that the request has been successfully recorded, and ‘0’ on failure. On a failure return, *func* will never be called. A failure indicates a (temporary) inability to allocate required internal data structures.

C.1.6 canput-test for room in a queue

```
int
canput(q)
    register queue_t *q;
```

canput(9) determines if there is room left in a message queue. If *q* does not have a **service** procedure, **canput(9)** will search further in the same direction in the *Stream* until it finds a queue containing a **service** procedure (this is the first queue on which the passed message can actually be enqueued). If such a queue cannot be found, the search terminates on the queue at the end of the *Stream*. **canput(9)** tests the queue found by the search. If the message queue in this queue is not full, **canput(9)** returns ‘1’. This return indicates that a message can be put to queue *q*. If the message queue is full, **canput(9)** returns ‘0’. In this case, the caller is generally referred to as blocked.

canput(9) only takes into account normal data flow control.

C.1.7 copyb-copy a message block

```
mblk_t *
copyb(bp)
    register mblk_t *bp;
```

copyb(9) copies the contents of the message block pointed at by *bp* into a newly-allocated message block of at least the same size. **copyb(9)** allocates a new block by calling **allocb(9)**. All data between the *b_rptr* and *b_wptr* pointers of a message block are copied to the new block, and these pointers in the new block are given the same offset values they had in the original message block.

On successful completion, **copyb(9)** returns a pointer to the new message block containing the copied data. Otherwise, it returns a ‘NULL’ pointer. The copy is rounded to full word boundary.

C.1.8 copymsg-copy a message

```
mbblk_t *
copymsg(bp)
    register mblk_t *bp;
```

`copymsg(9)` uses `copyb(9)` to copy the message blocks contained in the message pointed at by `bp` to newly allocated message blocks, and links the new message blocks to form the new message.

On successful completion, `copymsg(9)` returns a pointer to the new message. Otherwise, it returns a 'NULL' pointer.

C.1.9 datamsg-test whether message is a data message

```
#define datamsg(type) ((type) == M_DATA || (type) == M_PROTO \
    || (type) == M_PCPROTO || (type) == M_DELAY)
```

The `datamsg(9)` macro returns TRUE if '`mp->b_datap->db_type`' (where `mp` is declared as '`mbblk_t *mp`') is a data type message (that is, not a control message). In this case, a data type is `M_DATA`, `M_PROTO`, `M_PCPROTO`, or `M_DELAY`. If '`mp->b_datap->db_type`' is any other message type, `datamsg(9)` returns FALSE.

C.1.10 dupb-duplicate a message block descriptor

```
mbblk_t *
dupb(bp)
    mblk_t *bp;
```

`dupb(9)` duplicates the message block descriptor (`mbblk_t`) pointed at by `bp` by copying it into a newly allocated message block descriptor. A message block is formed with the new message block descriptor pointing to the same data block as the original descriptor. The reference count in the data block descriptor (`dbblk_t`) is incremented. `dupb(9)` does not copy the data buffer, only the message block descriptor.

On successful completion, `dupb(9)` returns a pointer to the new message block. If `dupb(9)` cannot allocate a new message block descriptor, it returns 'NULL'.

This routine allows message blocks that exist on different queues to reference the same data block. In general, if the contents of a message block with a reference count greater than '1' are to be modified, `copymsg(9)` should be used to create a new message block and only the new message block should be modified. This insures that other references to the original message block are not invalidated by unwanted changes.

C.1.11 dupmsg-duplicate a message

```
mbblk_t *
dupmsg(bp)
    mblk_t *bp;
```

`dupmsg(9)` calls `dupb(9)` to duplicate the message pointed at by `bp`, by copying all individual message block descriptors, and then linking the new message blocks to form the new message. `dupmsg(9)` does not copy data buffers, only message block descriptors.

On successful completion, `dupmsg(9)` returns a pointer to the new message. Otherwise, it returns 'NULL'.

C.1.12 enableok-re-allow a queue to be scheduled for service

```
void
enableok(q)
    queue_t *q;
```

enableok(9) cancels the effect of an earlier **noenable(9)** on the same queue *q*. It allows a queue to be scheduled for service that had previously been excluded from queue service by a call to **noenable(9)**.

C.1.13 esballoc-allocate message and data blocks

```
mblk_t *
esballoc(base, size, pri, fr_rtn)
    unsigned char *base;
    int size, pri;
    frtn_t *fr_rtn;
```

esballoc(9) allocates message and data blocks that point directly to a client-supplied buffer. **esballoc(9)** sets *db_base*, *b_rptr*, and *b_wptra* fields to *base* (data buffer start) and *db_lim* to ‘*base + size*’. The pointer to struct **free_rtn** is placed in the *db_freep* field of the data block.

The success of **esballoc(9)** depends on the success of **allocb(9)** and that *base*, *size*, and *fr_rtn* are not ‘NULL’, in which case **esballoc(9)** returns a pointer to a message block. If an error occurs, **esballoc(9)** returns ‘NULL’.

C.1.14 flushband-flush the messages in a given priority band

```
void
flushband(q, pri, flag)
    register queue_t *q;
    unsigned char pri;
    int flag;
```

flushband(9) provides modules and drivers with the capability to flush the messages associated in a given priority band. *flag* is defined the same as in **flushq(9)**. If *pri* is zero, only ordinary messages are flushed. Otherwise, messages are flushed from the band specified by *pri* according to the value of *flag*.

C.1.15 flushq-flush a queue

```
void
flushq(q, flag)
    register queue_t *q;
    int flag;
```

flushq(9) removes messages from the message queue in queue *q* and frees them, using **freemsg(9)**. If *flag* is set to **FLUSHDATA**, **flushq(9)** discards all **M_DATA**, **M_PROTO**, **M_PCPROTO**, and **M_DELAY** messages, but leaves all other messages on the queue. If *flag* is set to **FLUSHALL**, all messages are removed from the message queue and freed. **FLUSHALL** and **FLUSHDATA** are defined in ‘*sys/stream.h*’.

If a queue behind *q* is blocked, **flushq(9)** may enable the blocked queue, as described in **putq(9)**.

C.1.16 freeb-free a single message block

```
void
freeb(bp)
    struct msgb *bp;
```

freeb(9) will free (deallocate) the message block descriptor pointed at by *bp*, and free the corresponding data block if the reference count [see **dupb(9)**] in the data block descriptor (**data** structure) is equal to '1'. If the reference count is greater than '1', **freeb(9)** will not free the data block, but will decrement the reference count.

If the reference count is '1' and if the message was allocated by **esballoc(9)**, the function specified by the '**db_frtnp->free_func**' pointer is called with the parameter specified by '**db_frtnp->free_arg**'.

freeb(9) can't be used to free a multi-block message [see **freemsg(9)**]. Note that results will be unpredictable if the **freeb(9)** is called with a null argument. One should always check that pointer is non-*NULL* before using **freeb(9)**.

C.1.17 freemsg-free all message blocks in a message

```
void
freemsg(bp)
    mblk_t *bp;
```

freemsg(9) uses **freeb(9)** to free all message blocks and their corresponding data blocks for the message pointed at by *bp*.

C.1.18 getadmin()-return the pointer to the module

```
int
(*getadmin(mid)) ()
    ushort mid;
```

getadmin(9) returns the *qadmin* pointer to the module identified by *mid*. It returns 'NULL' on error.

C.1.19 getmid-return a module id

```
ushort
getmid(name)
    char *name;
```

getmid(9) returns the module id for the module identified by *name*. It returns '0' on error.

C.1.20 getq-get a message from a queue

```
mblk_t *
getq(q)
    register queue_t *q;
```

getq(9) gets the next available message from the queue pointed at by *q*. **getq(9)** returns a pointer to the message and removes that message from the queue. If no message is queued, **getq(9)** returns 'NULL'.

getq(9), and certain other utility routines, affect flow control in the *Stream* as follows: If **getq(9)** returns 'NULL', the queue is marked with **QWANTR** so that the next time a message is placed on it, it will be scheduled for service [enabled, see **qenable(9)**]. If the data

in the enqueued messages in the queue drop below the low water mark, *q_lowat*, and a queue behind the current queue had previously attempted to place a message in the queue and failed [i.e., was blocked, see `canput(9)`], then the queue behind the current queue is scheduled for service.

The queue count is maintained on a per-band basis. Priority band 0 (normal messages) uses *q_count*, *q_lowat*, etc. Nonzero priority bands use the fields in their respective `qband(9)` structures (*qb_count*, *qb_lowat*, etc). All messages appear on the same list, linked via their *b_next* pointers. *q_count* does not reflect the size of all messages on the queue; it only reflects those messages in the normal band of flow.

C.1.21 `insq`-put a message at a specific place in a queue

```
int
insq(q, emp, mp)
    register queue_t *q;
    register mblk_t *emp;
    register mblk_t *mp;
```

`insq(9)` places the message pointed at by *mp* in the message queue contained in the queue pointed at by *q* immediately before the already enqueued message pointed at by *emp*. If *emp* is 'NULL', the message is placed at the end of the queue. If *emp* is non-'NULL', it must point to a message that exists on the queue *q*, or a system panic could result.

If an attempt is made to insert a message out of order in a queue via `insq(9)`, the message will not be inserted and the routine fails.

The queue class of the new message is ignored. However, the priority band of the new message must adhere to the following ordering:

```
emp->b_prev->b_band >= mp->b_band >= emp->b_band.
```

This routine returns '1' on success and '0' on failure.

C.1.22 `linkb`-concatenate two messages into one

```
void
linkb(mp, bp)
    register mblk_t *mp;
    register mblk_t *bp;
```

`linkb(9)` puts the message pointed at by *bp* at the tail of the message pointed at by *mp*.

C.1.23 `msgdsize`-get the number of data bytes in a message

```
int
msgdsize(bp)
    register mblk_t *bp;
```

`msgdsize(9)` returns the number of bytes of data in the message pointed at by *bp*. Only bytes included in data blocks of type `M_DATA` are included in the total.

C.1.24 `noenable`-prevent a queue from being scheduled

```
void
noenable(q)
    queue_t *q;
```

noenable(9) prevents the queue *q* from being scheduled for service by **putq(9)** or **putbq(9)** when these routines enqueue an ordinary priority message, or by **insq(9)** when it enqueues any message. **noenable(9)** does not prevent the scheduling of queues when a high priority message is enqueued, unless it is enqueued by **insq(9)**.

C.1.25 OTHERQ-get pointer to the mate queue

```
#define OTHERQ(q) ((q)->q_flag & QREADR ? (q)+1 : (q)-1)
```

The **OTHERQ(9)** macro returns a pointer to the mate queue of *q*.

If *q* is the read queue for the module, it returns a pointer to the module's write queue. If *q* is the write queue for the module, it returns a pointer to the read queue.

C.1.26 pullupmsg-concatenate and align bytes in a message

```
int
pullupmsg(mp, len)
    struct msgb *mp;
    register int len;
```

pullupmsg(9) concatenates and aligns the first *len* data bytes of the passed message into a single, contiguous message block. Proper alignment is hardware-dependent. **pullupmsg(9)** only concatenates across message blocks of similar type. It fails if *mp* points to a message of less than *len* bytes of similar type. If *len* is '-1' **pullupmsg(9)** concatenates all the like-type blocks in the beginning of the message pointed at by *mp*.

On success, **pullupmsg(9)** returns '1' and, as a result of the concatenation, it may have altered the contents of the message pointed to by *mp*. On failure, it returns '0'.

C.1.27 putbq-return a message to the beginning of a queue

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

putbq(9) puts the message pointed at by *bp* at the beginning of the queue pointed at by *q*, in a position in accordance with the message type. High priority messages are placed at the head of the queue, followed by priority band messages and ordinary messages. Ordinary messages are placed after all high priority and priority band messages, but before all other ordinary messages already in a queue. The queue will be scheduled in accordance with the same rules described in **putq(9)**. This utility is typically used to replace a message on a queue from which it was just removed.

A service procedure must never put a high priority message back on its own queue, as this would result in an infinitive loop.

putbq(9) returns '1' on success and '0' on failure.

C.1.28 putctl-put a control message

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


On `putctl(9)` creates a control message of type `type`, and calls the put procedure of the queue pointed at by `q`, with a pointer to the created message as an argument. `putctl(9)` allocates new blocks by calling `allocb(9)`.

On successful completion, `putctl(9)` returns '1'. It returns '0', if it cannot allocate a message block, or if type `M_DATA`, `M_PROTO`, `M_PCPROTO`, or `M_DELAY` was specified.

C.1.29 `putctl1-put` a control message with a one-byte parameter

```
int
putctl1(q, type, param)
    queue_t *q;
```

`putctl1(9)` creates a control message of type `type` with a one-byte parameter `param`, and calls the put procedure of the queue pointed at by `q`, with a pointer to the created message as an argument. `putctl1(9)` allocates new blocks by calling `allocb(9)`.

On successful completion, `putctl1(9)` returns '1'. It returns '0', if it cannot allocate a message block, or if type `M_DATA`, `M_PROTO`, or `M_PCPROTO` was specified. `M_DELAY` is allowed.¹

C.1.30 `putnext-put` a message to the next queue

```
#define putnext(q, mp) ((*q)->q_next->q_qinfo->q_i_putq)((q)->q_next, (mp))
```

The `putnext(9)` macro calls the put procedure of the next queue in a *Stream* and passes it a message pointer as an argument. `q` is the calling queue (not the next queue) and `mp` is the message to be passed. `putnext(9)` is the typical means of passing messages to the next queue in a *Stream*.

C.1.31 `putq-put` a message on a queue

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

`putq(9)` puts the message pointed at by `bp` on the message queue contained in the queue pointed at by `q` and enables that queue. `putq(9)` queues messages based on message queueing priority. priority classes are high priority ('`type >= QPCTL`'), priority band ('`type < QPCTL && band > 0`'), and normal ('`type < QPCTL && band == 0`').

`putq(9)` always enables the queue when a high priority message is queued. `putq(9)` is allowed to enable the queue (`QNOENAB` is not set) if the message is the priority band message, or the `QWANTR` flag is set indicating that the `service` procedure is ready to read the queue. Note that the `service` procedure must never put a priority message back on its own queue, as this would result in an infinite loop. `putq(9)` enables the queue when an ordinary message is queued if the following condition is set, and enabling is not inhibited by `noenable(9)`: the condition is set if the module has just been pushed, or if no message was queued on the last `getq(9)` call, and no message has been queued since.

¹ This might be the answer to the `M_DELAY` mystery: it looks as though `M_DELAY` is acceptable for `putctl1(9)` but not for `putctl(9)`.

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

`putq(9)` is intended to be used from the `put` procedure in the same queue in which the message will be queued. A module should not call `putq(9)` directly to pass messages to a neighboring module. `putq(9)` may be used as the `qi_putp()` `put` procedure value in either or both of a module's `qinit(9)` structures. This effectively bypasses any `put` procedure processing and uses only the module's `service` procedure(s).

`putq(9)` returns '1' on success and '0' on failure.

C.1.32 qenable-enable a queue

```
void
qenable(q)
    register queue_t *q;
```

`qenable(9)` places the queue pointed at by `q` on the linked list of queues that are ready to be called by the *STREAMS* scheduler.

C.1.33 qreply-send a message on a *Stream* in the reverse direction

```
void
qreply(q, bp)
    register queue_t *q;
    mblk_t *bp;
```

`qreply(9)` sends the message pointed at by `bp` up (or down) the *Stream* in the reverse direction from the queue pointed at by `q`. This is done by locating the partner of `q` [see `OTHERQ(9)`], and then calling the `put` procedure of that queue's neighbor [as in `putnext(9)`]. `qreply(9)` is typically used to send back a response (`M_IOCACK` or `M_IOCNAK` message) to an `M_IOCTL` message.

C.1.34 qsize-find the number of messages on a queue

```
int
qsize(qp)
    register queue_t *qp;
```

`qsize(9)` returns the number of messages present in queue `qp`. If there are no messages on the queue, `qsize(9)` returns '0'.

C.1.35 RD-get pointer to the read queue

```
#define RD(q) ((q)-1)
```

The `RD(9)` macro accepts a write queue pointer, `q`, as an argument and returns a pointer to the read queue for the same module.

C.1.36 rmvb-remove a message block from a message

```
mbblk_t *
```

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

`rmvb(9)` removes the message block pointed at by `bp` from the message pointed at by `mp`, and then restores the linkage of the message blocks remaining in the message. `rmvb(9)` does not free the removed message block. `rmvb(9)` returns a pointer to the head of the resulting message. If `bp` is not contained in `mp`, `rmvb(9)` returns a ‘-1’. If there are no message blocks in the resulting message, `rmvb(9)` returns a ‘NULL’ pointer.

C.1.37 rmvq-remove a message from a queue

```
void
rmvq(q, mp)
    register queue_t *q;
    register mblk_t *mp;
```

`rmvq(9)` removes the message pointed at by `mp` from the message queue in the queue pointed at by `q`, and then restores the linkage of the messages remaining on the queue. If `mp` does not point to a message that is present on the queue `q`, a system panic could result.

C.1.38 splstr-set processor level

`splstr(9)` increases the system processor level to block interrupts at a level appropriate for *STREAMS* modules and drivers when they are executing critical portions of their code. `splstr(9)` returns the processor level at the time of its invocation. Module developers are expected to use the standard kernel function `splx(9)`, where the argument `s` is the integer value returned by `splstr(9)`, to restore the processor level to its previous value after the critical portions of code are passed.²

C.1.39 strlog-submit messages for logging

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

`strlog(9)` submits messages containing specified information to the `log(7)` driver. Required definitions are contained in ‘`sys/strlog.h`’ and ‘`sys/log.h`’. `mid` is the *STREAMS* module id number for the module or driver submitting the log message. `sid` is an internal sub-id number usually used to identify a particular minor device of a driver. `level` is a tracing level that allows selective screening of messages from the tracer. `flags` are any combination of:

- `SL_ERROR` (the message is for the error logger),
- `SL_TRACE` (the message is for the tracer),
- `SL_FATAL` (advisory notification of a fatal error),

² `splstr(9)` alone is unsuitable for use on *MP* system. Use basic locks instead.

- `SL_NOTIFY` (request that a copy of the message be mailed to the system administrator) (Note that `SL_NOTIFY` is not an option by itself, but rather a modifier to the `SL_ERROR` flag),
- `SL_CONSOLE` (log the message to the console),
- `SL_WARN` (warning message), and
- `SL_NOTE` (notice the message).

fmt is a `printf(3)` style format string, except that `'%s'`, `'%e'`, `'%E'`, `'%g'`, and `'%G'` conversion specifications are not handled. Up to `NLOGARGS` numeric or character arguments can be provided. [See `log(7)`.]

C.1.40 `strqget`-obtain information about a queue or band of the queue

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

`strqget(9)` allows modules and drivers to get information about a queue or particular band of the queue. The information is returned in the long referenced by *valp*. The fields that can be obtained are defined by the following:

```
typedef enum qfields {
    QHIWAT = 0,
    QLOWAT = 1,
    QMAXPSZ = 2,
    QMINPSZ = 3,
    QCOUNT = 4,
    QFIRST = 5,
    QLAST = 6,
    QFLAG = 7,
    QBAD = 8
} qfields_t;
```

`strqget(9)` returns `'0'` on success and an error number on failure.

C.1.41 `strqset`-change information about a queue or band of the queue

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

`strqset(9)` allows modules and drivers to change information about a queue or particular band of the queue. The updated information is provided by *val*. This routine returns `'0'` on success and an error number on failure. If the field is intended to be read-only, then the error `[EPERM]` is returned and the field is left unchanged.

C.1.42 testb-check for an available buffer

```
int
testb(size, pri)
    register size;
    unit pri;
```

testb(9) checks for the availability of a message buffer of size *size* without actually retrieving the buffer. **testb(9)** returns ‘1’ if the buffer is available, and ‘0’ if no buffer is available. A successful return value from **testb(9)** does not guarantee that a subsequent **allocb(9)** call will succeed (e.g., in the case of an interrupt routine taking buffers).

pri is as described in **allocb(9)**.

C.1.43 unbufcall-cancel a bufcall request

```
void
unbufcall(id)
    register int id;
```

unbufcall(9) cancels a **bufcall(9)** request. *id* identifies an event returned by the **bufcall(9)** request.

C.1.44 unlinkb-remove a message block from the head of a message

```
mbblk_t *
unlinkb(bp)
    register mblk_t *bp;
```

unlinkb(9) removes the first message block pointed at by *bp* and returns a pointer to the head of the resulting message. **unlinkb(9)** returns a ‘NULL’ pointer if there are no more message blocks in the message.

C.1.45 WR-get pointer to the write queue

```
#define WR(q) ((q)+1)
```

The **WR(9)** macro accepts a read queue pointer, *q*, as an argument and returns a pointer to the write queue for the same module.

C.2 DKI Interface

With the *DKI* interface (see [\[<undefined>\], page <undefined>](#)), the following *STREAMS* utilities are implemented as functions: **datamsg(9)**, **OTHERQ(9)**, **putnext(9)**, **RD(9)**, **splstr(9)**, and **WR(9)**. ‘**sys/ddi.h**’ must be included after ‘**sys/stream.h**’ to get function definitions instead of the macros.

C.3 Utility Routine Summary

ROUTINE	DESCRIPTION
adjmsg	trim bytes in a message
allocb	allocate a message block
backq	get pointer to the queue behind a given queue
bcanput	test for flow control in a given priority band

bufcall	recover from failure of allocb
canput	test for room in a queue
copyb	copy a message block
copymsg	copy a message
datamsg	test whether message is a data message
dupb	duplicate a message block descriptor
dupmsg	duplicate a message
enableok	re-allow a queue to be scheduled for service
esballoc	allocate message and data blocks
flushband	flush messages in a given priority band
flushq	flush a queue
freeb	free a message block
freemsg	free all message blocks in a message
getadmin	return a pointer to a module
getmid	return the module id
getq	get a message from a queue
insq	put a message at a specific place in a queue
linkb	concatenate two messages into one
msgdsz	get the number of data bytes in a message
noenable	prevent a queue from being scheduled
OTHERQ	get pointer to the mate queue
pullupmsg	concatenate and align bytes in a message
putbq	return a message to the beginning of a queue
putctl	put a control message
putctl1	put a control message with a one-byte parameter
putnext	put a message to the next queue
putq	put a message on a queue
qenable	enable a queue
qreply	send a message on a <i>Stream</i> in the reverse direction
qsize	find the number of messages on a queue
RD	get pointer to the read queue
rmvb	remove a message block from a message
rmvq	remove a message from a queue
splstr	set processor level
strlog	submit messages for logging
strqget	obtain information on a queue or a band of the queue
strqset	change information on a queue or a band of the queue
testb	check for an available buffer
unbufcall	cancel bufcall request
unlinkb	remove a message block from the head of a message
WR	get pointer to the write queue

Appendix D STREAMS Debugging

D.1 Debugging

This appendix provides some tools to assist in debugging *STREAMS*-based applications.

The kernel routine `cmn_err(9)` allows printing of formatted strings on a system console. It displays a specified message on the console and/or stores it in the *putbuf* that is a circular array in the kernel and contains output from `cmn_err(9)`. Its format is:

```
#include <sys/cmn_err.h>
cmn_err(level, fmt, ARGS)
    int level;
    char *fmt;
    int ARGS;
```

where `level` can take the following values:

- `CE_CONT` may be used as simple `printf(3)`. It is used to continue another message or to display an informative message not associated with an error.
- `CE_NOTE` report system events. It is used to display a message preceded with ‘NOTICE:’ This message is used to report system events that do not necessarily require user action, but may interest the system administrator. For example, a sector on a disk needing to be accessed repeatedly before it can be accessed correctly might be such an event.
- `CE_WARN` system events that require user action. This is used to display a message preceded with ‘WARNING:’ This message is used to report system events that require immediate attention, such as those where if an action is not taken, the system may panic. For example, when a peripheral device does not initialize correctly, this level should be used.
- `CE_PANIC` system panic. This is used to display a message preceded with ‘PANIC:’ Drivers should specify this level only under the most severe conditions. A valid use of this level is when the system cannot continue to function. If the error is recoverable, not essential to continued system operation, do not panic the system. This level halts all processing.
- `CE_IPANIC` generates a system panic like `CE_PANIC`, but does not synchronize the file system before the panic. See `CE_PANIC` for more information.

fmt and *ARGS* are passed to the kernel routine `printf(3)` that runs at `splhi(9)` and should be used sparingly. If the first character of *fmt* begins with `!` (an exclamation point) output is directed to *putbuf*. *putbuf* can be accessed with the `crash(1M)` command. If the destination character begins with `^` (a caret) output goes to the console. If no destination character is specified, the message is directed to both the *putbuf* array and the console. `cmn_err(9)` appends each *fmt* with ‘\n’, except for the `CE_CONT` level, even when a message is sent to the *putbuf* array. *ARGS* specifies a set arguments passed when the message is displayed. Valid specifications are ‘%s’ (string), ‘%u’ (unsigned decimal), ‘%d’ (decimal), ‘%o’ (octal), and ‘%x’ (hexadecimal). `cmn_err(9)` does not accept length specifications in conversion specifications. For example, ‘%3d’ is ignored.

D.2 crash(1M) Command

The **crash(1M)** command is used to examine kernel structures interactively. It can be used on system dump and on active system.

The following lists crash functions related to STREAMS:

- **dbfree** print data block header free list.
- **dblock** print allocated Streams data block headers.
- **linkblk** print the linkblk table.
- **mbfree** print free Streams message block headers.
- **mblock** print allocated Streams message block headers.
- **pty** print pseudo ttys presently configured. The **-l** option gives information on the line discipline module **ldterm**, the **-h** option provides information on the pseudo-tty emulation module **ptem**, and the **-s** option gives information on the packet module **pckt**.
- **qrun** print a list of scheduled queues.
- **queue** print STREAMS queues.
- **stream** print the stdata table.
- **strstat** print STREAMS statistics.
- **tty** print the tty table. The **-l** option prints out details about the line discipline module.

The crash functions **dblock**, **linkblk**, **mblock**, **queue**, and **stream** take an optional table entry argument or address that is the address of the data structure. The **strstat** command gives information about *STREAMS* event cells and linkblks in addition to message blocks, data blocks, queues, and *Streams*. On the output report represents the number of structures currently configured. It may change because resources are allocated as needed.

The following example is a sample output from **crash(1M)**:

Output from **crash(1M)** may look different depending on the version is used. Examples in the section were produced using UXP/V, and they were also formatted for easier reference.

```
$ crash dump/dump stand/unix
dumpfile = dump/dump, namelist = stand/unix, outfile = stdout
>strstat
ITEM                CONFIG  ALLOC  FREE    TOTAL    MAX    FAIL
streams              102    102    0        826    108    0
queues               546    546    0       2689    576    0
message blocks       175     76    99     62390   399    0
data blocks          147     76    71     57265   368    0
link blocks           7       7     0         7     7     0
streams events        5       4     1         7     5     0

Count of scheduled queues: 0

>stream
STREAM TABLE SIZE = 102
STREAM TABLE SIZE = 102
ADDRESS      WRQ      IOCB      VNODE  PUSHCNT  RERR/WERR FLAG
7bff8900 7c083540  1706f0 7c468904      1 0/0  rslp mrd pdis spip
```



```

7c0c9e00 7c139840 182248 7c139e04 3 0/0 istty mnds ondel
7bffdb00 7c139c40 1822b4 7c139904 1 0/0 istty
7c0c8300 7c0cb640 1706f0 7c139d04 1 0/0 mrd pdis spip
7bff8a00 7c0ca240 182248 7c071604 3 0/0 rslp istty mrd ondel
7c0c8580 7c0ca640 1822b4 7c0ca304 0 0/0 istty
7c0c8e00 7c0cab40 1706f0 7c0ca704 1 0/0 mrd pdis spip

```

The following example illustrates debugging of a line printer. Knowledge of the data structures of the driver is needed for debugging. The example starts with the following data structure of the line printer driver:

```

struct lp {
    short lp_flags;
    queue_t *lp_qptr;          /* back pointer to write queue */
};
extern struct lp lp_lp[];

```

The first command ‘nm lp_lp’ prints the value and type for the line printer driver data structure. The second command ‘rd 40275750 20’ prints ‘20’ values starting from the location ‘40275750’ (note that the function rd is alias of od). The third command ‘size queue’ gives the size of the queue structure. The next two functions again give the ‘20’ values starting at the specified locations in the hexadecimal format. The command ‘rd -c 4032bf40 32’ gives the character representation of the value in the given location. The option ‘-x’ gives a value in the hexadecimal representation and the option ‘-a’ produces the same in the *ASCII* format.

```

$ /usr/sbin/crash
dumpfile - /dev/mem, namelist = /stand/unix, outfile = stdout

>nm lp_lp
lp_lp 40275750 bss

>rd 40275750 20
40275750: 00000000 00000000 00000000 40262f60
40275760: 00000000 00000000 00000000 00000000
40275770: 00000000 00000000 00000000 00000000
40275780: 00000000 00000000 00000000 00000000
40275790: 00000000 00000000 00000000 00000000

>size queue
36

>rd 40262f60 20
40262f60: 4017315c 402624a4 4026257c 00000000
40262f70: 00000000 40275758 0200002e 00000200
40262f80: 02000100 00000000 00000000 00000000
40262f90: 00000000 00000000 00000000 00000000
40262fa0: 00000000 00000000 00000000 00000000

>rd 402624a4 20
402624a4: 40262624 00000000 00000000 4032bf40
402624b4: 4032bf5f 40236884 4026233c 00000000
402624c4: 00000000 40331fd9 40331fd9 00000000
402624d4: 00000000 00000000 00000000 4032bf80
402624e4: 4032bf80 40236894 40262564 00000000

```

```

>rd -c 4032bf40 32
4032bf40: little red light
4032bf50:   on the highway

>rd -x 40262624 20
40262624: 40262594 402624a4 00000000 4032bd40
40262634: 4032bd5f 40236804 00000000 00000000
40262644: 00000000 4030c800 4030c800 402319e4
40262654: 00000000 00000000 00000000 4032be40
40262664: 4032be40 40236844 4026239c 00000000

>rd -a 4032bd40 31
little red light on the highway

```

D.3 Dump Module Example

The following dump module example represents only one way of debugging *STREAMS* modules and drivers; using the `strlog(9)` function is another way. `strlog(9)` is discussed later in the chapter.

The `dump` module (its creator calls it "primitive but handy at times when a driver is not working properly or some other anomalies occur") has advantages over logging messages in that it will print all data passing to and from the module in the order they are received. One can modify this module to print more detailed information on the particular types of messages (e.g., special `M_IOCTL` messages) that a user is interested in.

This `dump` module is useful for looking at the sequence of messages passing on a *Stream* and to know who is doing what and when. For example, if a user is faced with a situation where a module is not passing through some messages correctly and user processes are hung waiting the messages to be returned, this module may help diagnose the problem. Another example is a situation where `M_IOCTL` messages are causing problems in a driver. This module can help to pinpoint the messages and their sequence without going back to the source of the programs and trying to figure out what is happening in particular cases. This module is also useful in debugging inter-module communication protocols (e.g., `M_CTL` or `M_PROTO` between two cooperating modules).

This example should not be used as is for debugging in more than one place. However, it can be modified quite easily to print minor device numbers along with each message, so that it can be inserted in two places around a particular module for looking at both ends of the module simultaneously.

There are two disadvantages in this module approach: this module cannot be used to debug the console driver, and it drastically alters the timing characteristics of the machine and the *Stream* in which it is running. Therefore, this example module is not meant to be used for discovering timing-related problems such as interrupt timing and priority level changes.

The `dump` module is given here only as an illustration and a possible aid to developers in debugging their applications. First the appropriate header file is provided followed by the master file and the code.

```

/*
 * dump.h Header for DUMP module.
 */

```

```

#define DUMPIOC          ('Q' << 8) /* define to be unique */

#define DUMP_VERB (DUMPIOC | 1)
#define DUMP_TERSE (DUMPIOC | 2)

#define D_FREE    0          /* slot free */
#define D_USED    1          /* slot in use */

#define D_OUT     1          /* outgoing data */
#define D_IN      2          /* incoming data */

#define D_VERB    0x01       /* verbose option on */

struct dm_str {
    char dm_use;              /* non-zero if in use */
    char dm_state;            /* for state during console output */
    char dm_flags;            /* flags */
};

*
* DUMP - STREAMS message dump module
*
*FLAG #VEC PREFIX sort #DEV IPL DEPENDENCIES/VARIABLES
m    -    dump
                                dump_users[#C] (%c%c%c)
                                dm_ucnt (%i) = (#C)

/*
* DUMP module.  This module prints data and ioctls going to and
* from a device in real time.  Printout is on the console.  Usage
* is to push it into a Stream between any other modules.
*
* DUMP_VERB    Verbose printing of M_DATA (default)
* DUMP_THRSE    Terse printing of data (optional)
*
* The messages it prints begin with "I:" for incoming, "O:" for
* outgoing data.  "Ci" or "Co" are for non-data (control) messages.
* Data is printed in character or hexadecimal format delimited by ((
* and )) at message boundaries.
*/

#include <sys/types.h> /* required in all modules and
drivers */
#include <sys/stream.h> /* required in all modules and
drivers */
#include <sys/param.h>
#include <sys/fcntl.h>
#include <sys/cmn_err.h>
#include <dump.h> /* local ioctls */
#include <sys/termio.h>

static struct module_info dumprinfo =
    { 0x6475, "dump", 0, INFPSZ, 0, 0 };
static struct module_info dumpwinfo =
    { 0x6475, "dump", 0, INFPSZ, 0, 0 };
static int dumpopen(), dumprput(), dumpwput(), dumpclose();

```

```

static struct qinit rinit = {
    dumpprput, NULL, dumpopen, dumpclose, NULL, &dumpprinfo, NULL
};

static struct qinit winit = {
    dumpwput, NULL, NULL, NULL,, NULL, &dumpprinfo, NULL
};

struct streamtab dumpinfo = { &rinit, &winit, NULL, NULL };

extern int dm_ucnt; /* count of dm_users */

extern struct dm_str dm_users[]

/*
 * dumpopen          open us and turn us on.
 */
static int
dumpopen(q, dev, flag, sflag)
queue_t *q; /* pointer to read queue */
dev_t dev; /* major/minor device number */
int flag; /* file open flag */

int sflag; /* stream open flag */
{
    register int i;

    if (q->q_ptr != NULL) {
        cmn_err(CE_CONT, "^DUMP: re-open slot %d",
            ((struct dm_str *) q->q_ptr - dm_users));
        if (flag & O_NDELAY)
            cmn_err(CE_CONT, "^dump: re-open: O_NDELAY set 0");
        return 0;
    }
    for (i = 0; i < dm_ucnt; i++) {
        if (dm_users[i].dm_use == D_FREE) {
            dm_users[i].dm_use = D_USED;
            dm_users[i].dm_state = 0;
            dm_users[i].dm_flags = D_VERB;
            q->q_ptr = (caddr_t) &dm_users[i];
            wr(q)->q_ptr = (caddr_t) &dm_users[i];
            if (flag & O_NDELAY)
                cmn_err(CE_CONT, "^DUMP: open: O_NDELAY set 0");
            return 0;
        }
    }
    return OPENFAIL;
}

/*
 * dumpclose        Close us down.
 */
static int
dumpclose(q, flag)
queue_t *q; /* pointer to the read queue */
int flag; /* file flags */
{

```

```

    struct dm_str *d;

    d = (struct dm_str *) q->q_ptr;
    d->dm_use = D_FREE;
    d->dm_flags = 0;
    q->q_ptr = 0;
    wr(q)->q_ptr = 0;
    return;
}

/*
 * dumpwput      Put procedure for WRITE side of module.  Gathers
 *               data from all passing M_DATA messages.  Calls
 *               rioutine to handle ioctl calls.
 */
static int
dumpwput(q, mp)
queue_t *q; /* pointer to the write queue */
mblk_t *mp; /* message pointer */
{
    struct iocblk *iocp;
    struct dm_str *d;

    d = (struct dm_str *) q->q_ptr;
    if (mp->b_datap->db_type == M_IOCTL) {
        iocp = (struct iocblk *) mp->b_rptr;
        if ((iocp->ioc_cmd & DUMPIOC) == DUMPIOC) {
            dumpioc(q, mp);
            return;
        } else {
            cmn_err(CE_CONT, "^0o:M_IOCTL %x, cnt %d ",
                iocp->ioc_cmd, iocp->ioc_count);
            if ((d->dm_flags & D_VERB) && mp->b_cont)
                dumpshow(mp->b_cont, iocp->ioc_cmd);
            d->dm_state = 0;
            putnext(q, mp);
        }
    } else if (mp->b_datap->db_type == M_DATA) {
        dumpgather(q, mp, D_OUT);
        putnext(q, mp);
    } else {
        dumpctl(q, mp, d, D_OUT);
        d->dm_state = 0;
        putnext(q, mp); /* pass message through */
    }
}

/*
 * dumprput      Read side put procedure.  Snag all M_DATA
 *               messages.
 */
static int
dumprput(q, mp)
queue_t *q; /* pointer to the read queue */
mblk_t *mp; /* message pointer */
{
    struct dm_str *d;

```

```

        d = (struct dm_str *) q->q_ptr;
        if (mp->b_datap->db_type == M_DATA) {
dumpgather(q, mp, D_IN);

        } else {
dumpctl(q, mp, d, D_IN);
d->dm_state = 0;
        }
        putnext(q, mp); /* pass message through */
}

/*
 * dumpgather      Gather info from this data message and print it. We
 *                  don't "putnext", as that is done by the caller, in
 *                  the appropriate direction.
 */
dumpgather(q, mp, dir)
queue_t *q;
mbblk_t *mp;
int dir;
{
    register struct dm_str *d;
    register int sx;
    register unsigned char *readp;
    register mblk_t *tmp;
    int counter;
    char junk[2];

    d = (struct dm_str *) q->q_ptr;
    /*
     * when dumping to console, check state & print I/O if it
     * changes.
     */
    if (d->dm_state != dir) {
d->dm_state = dir;
cmn_err(CE_CONT, "^lls", ((dir == D_IN) ? "I:" : "O:"));
    }
    if ((!mp->b_datap) || ((mp->b_wptr) && (mp->b_cont == NULL))) {
/* Trap zero length messages going past! */
cmn_err(CE_CONT, "^DUMP: 0 len data msg %s. ",
(dir == D_OUT) ? "OUT" : "IN");
return;
    }
    cmn_err(CE_CONT, "^{");
    tmp = mp;
    counter = 0;
    sx = splstr();
    junk[1] = ' ';
    more:readp = tmp->b_rptr;
    while (readp < tmp->b_wptr) {

if (d->dm_flags & D_VERB) {
    if ((*readp >= ' ') && (*readp <= '~')
&& !(*READP & 0x80)) {
junk[0] = *readp;
cmn_err(CE_CONT, "%s", junk);

```

```

        } else
        cmn_err(CE_CONT, "~0x%x", *readp);
    } else {
        ++counter;
    }
    ++readp;
    }
    if ((tmp->b_cont) && (tmp->b_datap->db_type == M_DATA)) {
tmp = tmp->b_cont;
goto more;
    }
    if (!(d->dm_flags & D_VERB))
cmn_err(CE_CONT, "~%d", counter);
    cmn_err(CE_CONT, "~}~");
    if (tmp->b_cont && (tmp->b_datap->db_type != M_DATA))
cmn_err(CE_CONT, "~DUMP: non-data b_cont");
    splx(sx);
}

/*
 * dumpioc          Completely handle one of our ioctl calls, including
 *                  the greply of the message.
 */
dumpioc(q, mp)

queue_t *q;
mblk_t *mp;
{
    register struct iocblk *iocp;
    register struct dm_str *d, *ret;

    d = (struct dm_str *) q->q_ptr;
    iocp = (struct iocblk *) mb->b_datap->db_base;
    cmn_err(CE_CONT, "~DUMP: own ioctl is ");
    switch (iocp->ioc_cmd) {
        case DUMP_VERB:
d->dm_flags |= D_VERB;
cmn_err(CE_CONT, "~DUMP_VERB0");
break;

        case DUMP_TERSE:
d->dm_flags &= ~D_VERB;
cmn_err(CE_CONT, "~DUMP_TERSE0");
break;

        default:
mp->b_datap->db_type = M_IOCNAK;
cmd_err(CE_CONT, "~UNKNOWN DUMP IOCTL x%x0", iocp->ioc_cmd);
greply(q, mp);
return;
    }
    mp->b_datap->db_type = M_IOCACK;
    iocp->ioc_count = 0;
    greply(q, mp);
}

/*

```

```

    * dumpctl          Display information about control messages.
    */
dumpctl(q, mp, d, dir)
queue_t *q;
mblk_t *mp;
struct dm_str *d;
int dir;
{
    cmn_err(CE_CONT, "^O%s: M_", ((dir == D_IN) ? "i" : "o"));
    switch (mp->b_datap->db_type) {
        case M_DATA: /* just in case */
            cmn_err(CE_CONT, "^DATA");
            break;

        case M_READ:
            cmn_err(CE_CONT, "^READ");
            break;

        case M_IOCTL:
            cmn_err(CE_CONT, "^IOCTL");
            break;

        case M_IOCACK:
            cmn_err(CE_CONT, "^IOCACK");
            break;

        case M_IOCNAK:
            cmn_err(CE_CONT, "^IOCNAK");
            break;

        case M_IODATA:
            cmn_err(CE_CONT, "^IODATA");
            break;

        case M_CTL:
            cmn_err(CE_CONT, "^CTL");
            break;

        case M_PROTO:
            cmn_err(CE_CONT, "^PROTO");
            break;

        case M_PCPROTO:
            cmn_err(CE_CONT, "^PCPROTO");
            break;

        case M_BREAK:
            cmn_err(CE_CONT, "^BREAK");
            break;

        case M_DELAY:
            cmn_err(CE_CONT, "^DELAY");
            break;

        case M_PASSFP:
            cmn_err(CE_CONT, "^PASSFP");

```



```

break;

    case M_SETOPTS:
cmn_err(CE_CONT, "^SETOPTS");
break;

    case M_SIG:
cmn_err(CE_CONT, "^SIG");
cmn_err(CE_CONT, "^(%d) ", (int) *mp->b_rptr);
break;

    case M_ERROR:
cmn_err(CE_CONT, "^ERROR");
break;

    case M_HANGUP:
cmn_err(CE_CONT, "^HANGUP");
break;

    case M_FLUSH:
cmn_err(CE_CONT, "^FLUSH");
break;

    case M_PCSIG:
cmn_err(CE_CONT, "^PCSIG");
cmn_err(CE_CONT, "^(%d) ", (int) *mp->b_rptr);
break;

    case M_COPYOUT:
cmn_err(CE_CONT, "^COPYOUT");
break;

    case M_COPYIN:
cmn_err(CE_CONT, "^COPYIN");
break;

    case M_START:
cmn_err(CE_CONT, "^START");
break;

    case M_STOP:
cmn_err(CE_CONT, "^STOP");
break;

    case M_STARTI:
cmn_err(CE_CONT, "^STARTI");
break;

    case M_STOPI:
cmn_err(CE_CONT, "^STOPI");
break;

    default:
cmn_err(CE_CONT, "^Unknown! 07 07");
}
}

```

```

/*
 * dumpshow          Display information about known ioctls.
 */
dumpshow(mp, cmd)
mblk_t *mp; /* pointer to cont block of ioctl
message */
int cmd; /* ioc_cmd field */
{
    int i;
    struct timeio *t;

    /*
     * This is an example of printing data from IOCTL messages that
     * we're interested in. Add others as needed.
     */
    switch (cmd) {
        case TCSETAF:
            cmn_err(CE_CONT, "^TCSETAF ");
            goto prtall;

        case TCSETA:
            cmn_err(CE_CONT, "^TCSETA ");
            goto prtall;

        case TCSETAW:
            cmn_err(CE_CONT, "^TCSETAW ");
            prtall:
            t = (struct termio *) mp->b_rptr;
            cmn_err(CE_CONT, "if=%x ; of=%x ; cf=%x ; lf=%x0_cc=",
t->c_iflag, t->c_oflag, t->c_cflag, t->c_lflag);
            for (i = 0; i < NCC; i++)
                cmn_err(CE_CONT, "^0x%x ", (int) t->c_cc[1]);
            cmn_err(CE_CONT, "^0");
            break;
        default:
            return;
    }
}

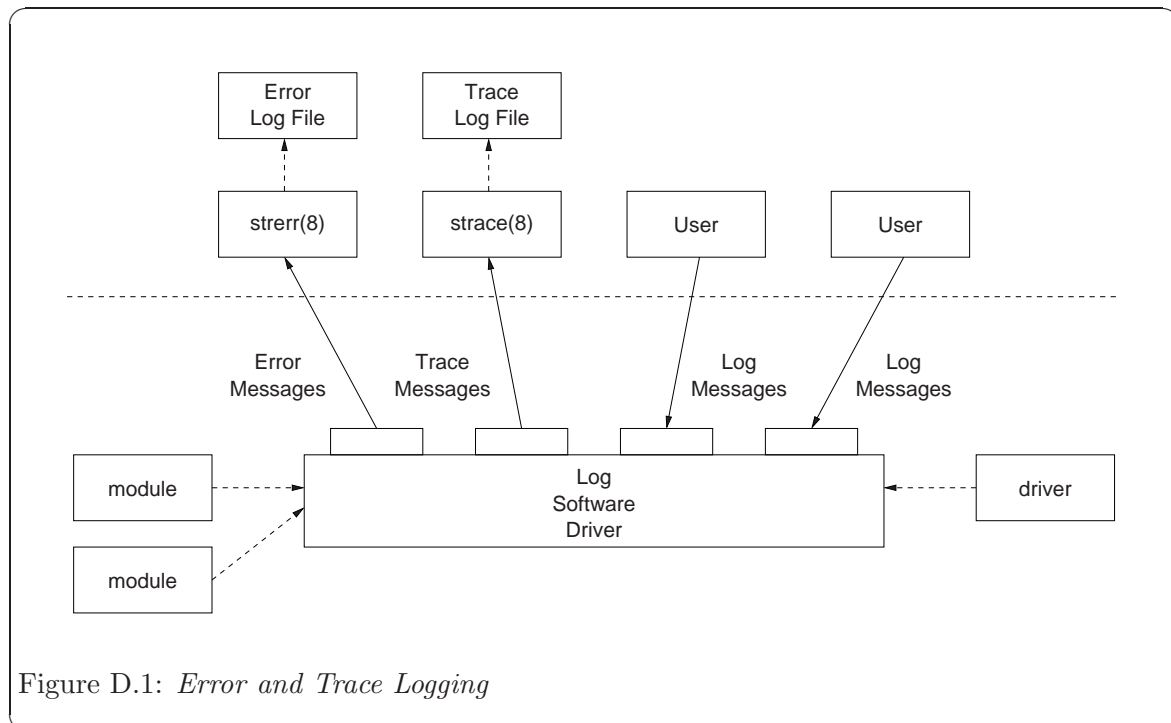
```

The situation with "cooperating modules" that seems to contradict the basic idea of reusability and independence of modules has risen in the internationalization context (see [\[undefined\]](#), page [\[undefined\]](#)) related to input methods of languages. It turns out that some back-end processing for input methods can be re-used and the size of the kernel reduced from what would be necessary by having two different full modules when only front-end processing needs to be different for different input methods. In this situation, one will save as much space as the size of the back-end module. The way this relates to the `dump` module is that this `dump` module can be used to help debug the protocol between the front-end and back-end modules. (The protocol is standard across various front-ends.)

D.4 Error and Trace Logging

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

Any module or driver in any *Stream* can call the *STREAMS* logging function `strlog(9)`, described in `log(7)`. `strlog(9)` is also described in [Appendix C \[STREAMS Utilities\]](#), [page 295](#). When called, `strlog(9)` will send formatted text to the error logger `strerr(1)`, the trace logger `strace(1)`, or the console logger.



`strerr(1)` is intended to operate as a daemon process initiated at system startup. A call to `strlog(9)` requesting an error to be logged causes an `M_PROTO` message to be sent to `strerr(1)`, which formats the contents and places them in a daily file. The utility `strclean(1)` is provided to purge daily log files that have not been modified for 3 days.

A call to `strlog(9)` requesting trace information to be logged causes a similar `M_PROTO` message to be sent to `strace(1)`, which places it in a user designated file. `strace(1)` is intended to be initiated by a user. The user can designate the modules/drivers and severity level of the messages to be accepted for logging by `strace(1)`.

A user process can submit its own `M_PROTO` messages to the `log(4)` driver for inclusion in the logger of its choice through `putmsg(2)`. The messages must be in the same format required by the logging processes and will be switched to the logger(s) requested in the message.

The output to the log files is formatted, *ASCII* text. The files can be processed by standard system commands such as `grep(1)` or `ed(1)`, or by developer-provided routines.

Appendix E STREAMS Configuration

E.1 Configuration

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

This appendix also includes a list of STREAMS related tunable parameters and describes the autopush facility.

E.1.1 Configuring STREAMS Modules and Drivers

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

The distinction is contained in the d_str field of the cdevsw structure. The d_str field provides the appropriate single entry point for all system calls on STREAMS files, as shown below:

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

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

```
struct streamtab prefixinfo = {...
```

If the d_str entry contains a non-null pointer, the operating system will recognize the device as a STREAMS driver and will call the appropriate STREAMS routine. If the entry is null, a traditional character I/O device cdevsw interface is used. NOTE that only the streamtab structure must be externally defined in STREAMS drivers and modules. The streamtab is used to identify the appropriate open, close, put, service, and administration routines. These driver and module routines should generally be declared static.

The configuration mechanism supports various combinations of block, character, STREAMS devices, and STREAMS modules. For example, it is possible to identify a device as a block and STREAMS device (although this is very unlikely), and entries will be inserted in the appropriate system switch tables.

When a STREAMS module is configured, an fmodsw table entry is generated by the configuration mechanism. The fmodsw contains the following:

```

#define FMNAMESZ 8

extern struct fmodsw {
    char    f_name[FMNAMESZ+1];
    struct  streamtab *f_str;
    int     *f_flag; /* same as device flag */
} fmodsw[];

```

where `f_name` is the name of the module used in STREAMS related ioctl calls. `f_str` is similar to the `d_str` entry in the `cdevsw` table. It is a pointer to a `streamtab` structure which contains pointers to the `qinit` structures for the read and write queues of this STREAMS module (as in STREAMS drivers). The module must contain the external definition:

```
struct streamtab prefixinfo = { ...
```

E.1.1.1 Configuration Examples

This section shows examples of configuring the following STREAMS driver and module:

loop the STREAMS loop-around software driver of Drivers

crmod the conversion module of Modules

To configure the STREAMS software (pseudo device) driver, variables, the following must appear in the master file.

```

* LOOP - STREAMS loop-around software driver
*
*FLAG #VEC PREFIX SOFT #DEV IPL DEPENDENCIES/VARIABLES
fs    -    loop   62    -    -
                                loop_loop[NLP ](%i%i)
                                loop_cnt (%i) = {NLP }

$$$$
NLP = 2

```

The flag field is set to "fs" which signifies that it is a STREAMS driver and a software driver. The prefix "loop" requires that the `streamtab` structure for the driver be defined as `loopinfo`. "62" is an arbitrary, software driver major number. The `#dev` field is "-" which shows that there are no sub-devices. The next field, dependencies/variables, is optional and contains a list of other drivers and modules that must be present in the configuration of this driver.

To configure the STREAMS module `crmod`, the following must appear in the `chmod` master file.

```

* CRMOD stream conversion module
*
*FLAG #VEC PREFIX SOFT #DEV IPL DEPENDENCIES/VARIABLES
m     -    crmd

```

The flag field is set to "m" which signifies that it is a STREAMS module. The prefix "crmd" requires that the `streamtab` structure for the module be defined as `crmdinfo`. The configuration mechanism uses the name of the master file (`crmod` in this case) to create the module name field (`f_name`) of the associated `fmodsw` entry. The prefix and module name can be different.

Neither of the given examples are hardware drivers. Configuring a STREAMS hardware driver is similar to configuring a character I/O hardware driver.

At times, it is useful to make a module both a module and a driver. There are not many good reasons to do so, but in some cases it's a good way to solve odd problems. It is done in the following way:

We called our module-driver combination thing in this description. First, the module open routine is handled differently. We also need to keep the state flag `sflag` that tells for any particular instantiation whether thing is to behave as a module or a driver, because `ioctl` handling is usually different depending on whether the thing is a driver or a module. Then, we need to make sure that the flags are set properly in the master file. Here is a sample master file for thing:

```
*
*FLAG #VEC PREFIX SOFT #DEV IPL DEPENDENCIES/VARIABLES
fsm - can 52 - - canstr[#C] (%0x40)
can_cnt (%i) = {#C}
$$$$
```

Notice the flag field contains "f" for being a Streams driver, "s" for being a software driver, and "m" for being a Streams module. We need to make sure that the module has a character device node in the file system, either as a cloneable driver or a regular driver. Once we have rebooted our system, thing can be opened by its name.

There is a trick in the open routine where `sflag` is checked:

```
.
.
.
if (sflag == MODOPEN)
    /* then the module is being pushed */

else if (sflag == CLONEOPEN)
    /* then its being opened as a cloneable driver */

else
    /* its being opened as a regular driver */
.
.
.
```

File systems that support STREAMS devices go through `cdevsw` to get the driver open routine. If `d_str` is set, then thing is a STREAMS driver, and it goes to the `fmodsw` to get the open routine. When one pushes a Streams module, the push code also goes to the `fmodsw` looking for the module by name. Depending on which way the open routine is called the `sflag` argument will be `MODOPEN`, `CLONEOPEN`, or zero.

E.1.1.2 Tunable Parameters

Certain system parameters referenced by STREAMS are configurable when a new operating system is being built. (Refer to the System Administrator's Guide for more information.). These parameters are:

- NSTRPUSH** Maximum number (should be at least 8) of modules that may be pushed onto a single Stream.
- STRMSGSZ** Maximum number of bytes of information that a single system call can pass to a Stream to be placed into the data part of a message (in `M_DATA` blocks). Any

write(2) exceeding this size will be broken into multiple messages. A putmsg(2) with a data part exceeding this size will fail with ERANGE. If STRMSGSZ is set to 0, then the number of bytes passed to a Stream is effectively infinite.

STRCTLSZ Maximum number of bytes of information that a single system call can pass to a Stream to be placed into the control part of a message (in an M_PROTO or M_PCPROTO block). A putmsg(2) with a control part exceeding this size will fail with ERANGE.

E.1.2 Autopush Facility

The Autopush facility [see autopush(1M)] is a general mechanism that configures the list of modules for a STREAMS device. It automatically pushes a pre-specified list of modules onto the Stream when the STREAMS device is opened and the device is not already open. The STREAMS Administrative Driver (SAD) [see sad(7)] provides an interface to the autopush mechanism. System administrators can open the SAD driver and set or get autopush information on other drivers. The SAD driver caches the list of modules to push for each driver. When the driver is opened, if not already open, the Stream head checks the SAD's cache to see if the device opened has been configured to have modules pushed automatically. If an entry is found, the modules are pushed. If the device has already been opened but has not yet been closed, another open would not cause the list of the pre-specified modules to be pushed again.

Three options are available to configure the module list:

- Configure for each minor device that is, a specific major and minor device number.
- Configure for a range of minor devices within a major device.
- Configure for all minor devices within a major device.

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

E.1.2.1 User Interface

The SAD driver can be accessed via the node /dev/sad/admin or /dev/sad/user. After the device is initialized, a program can be run to perform any needed autopush configuration. The program should open the SAD driver, read a configuration file to find out what modules are needed to be configured for which devices, format the information into strapush structures, and perform the necessary SAD_SAP ioctls.

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

The ioctl is a form of ioctl(fd, cmd, arg), where fd is the file descriptor of the SAD driver, cmd is either SAD_SAP (set autopush information) or SAD_GAP (get autopush information), and arg is a pointer to the structure strapush.

The structure strapush is defined as:

```
/*
 * maximum number fo modules that can be pushed on a Stream using the
```



```

    * autopush feature should be less than NSTRPUSH
    */
#define MAXAPUSH 8

    /* autopush information common to user and kernel */

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

    /* ap_cmd - various options of autopush */

#define SAP_CLEAR 0 /* remove configuration list */
#define SAP_ONE 1 /* configure one minor device */
#define SAP_RANGE 2 /* configure range of minor device */
#define SAP_ALL 3 /* configure all minor devices */

    /* format of autopush ioctls */

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

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

```

A device is identified by its major device number, `sap_major`. The `SAD_SAP` ioctl (`sap_cmd`) can take the following options:

- `SAP_ONE` configures a single minor device, `sap_minor`, of a driver.
- `SAP_RANGE` configures a range of minor devices from `sap_minor` to `sap_lastminor`, inclusive.
- `SAP_ALL` configures all minor devices of a device.
- `SAP_CLEAR` clears the previous settings by removing the entry with the matching `sap_major` and `sap_minor` fields.

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

A user may query the current configuration status of a given major/minor device by issuing the `SAD_GAP` ioctl with `sap_major` and `sap_minor` values of the device set. On successful return from this system call, the `strapush` structure will be filled in with the corresponding information for that device.

The following is an example of an autopush configuration file:

Appendix E: STREAMS Configuration

```
21 5 0 mod1 mod2      # configure a single minor device
22 -1 0 mod0 mod5 mod9 # configure all minor devices for major 22
39 3 18 mod7          # configure a range of minor devices
```

The first line represents the configuration for a single minor device whose major and minor numbers are 21 and 5 respectively. Two modules, mod1 and mod2, are automatically pushed on the Stream for this minor device. mod1 is pushed first, and mod2 is pushed next. The second line represents the configuration for all minor devices whose major number is 22. Three modules, mod0, mod5, and mod9, are pushed automatically on the Stream. The last line represents the configuration for the range of minor devices from 3 to 18 whose major device number is 39. Only the module, mod7, is pushed with this configuration.

The maximum number of entries the SAD driver can cache is determined by the tunable parameter NAUTOPUSH found in the SAD driver's master file.

Appendix F Conformance

F.1 SVR 4.2 MP DDI/DKI Compatibility

F.2 AIX 5L Version 5.1 Compatibility

F.3 HP-UX 11.0i v2 Compatibility

F.4 OSF/1 1.2/Digital UNIX Compatibility

F.5 UnixWare 7.1.3 Compatibility

F.6 Solaris 9/SunOS 5.9 Compatibility

F.7 Super/UX Compatibility

F.8 UXP/V Compatibility

F.9 LiS 2.18.1 Compatibility

Appendix G Portability

Although each of the manual pages of supported functions and structures provides compatibility and porting information, this document attempts to gather together pertinent information concerning porting from various *UNIX* operating system supporting *STREAMS*.

The porting information is organized by the operating system from which porting is being attempted. Note that, aside from configuration details, any system not listed here that is based on *SVR 4.2 MP* or on another of the implementations, should start with that implementation's portability information.

Porting information is organized into sections as follows:

G.1 Porting with Core Function Support

G.1.1 Core Message Functions

<code>adjmsg(9)</code>	trim bytes from the front or back of a <i>STREAMS</i> message
<code>allocb(9)</code>	allocate a <i>STREAMS</i> message and data block
<code>bufcall(9)</code>	install a buffer callback
<code>copyb(9)</code>	copy a <i>STREAMS</i> message block
<code>copymsg(9)</code>	copy a <i>STREAMS</i> message
<code>datamsg(9)</code>	tests a <i>STREAMS</i> message type for data
<code>dupb(9)</code>	duplicate a <i>STREAMS</i> message block
<code>dupmsg(9)</code>	duplicate a <i>STREAMS</i> message
<code>esballoc(9)</code>	allocate a <i>STREAMS</i> message and data block with a caller supplied data buffer
<code>freeb(9)</code>	frees a <i>STREAMS</i> message block
<code>freemsg(9)</code>	frees a <i>STREAMS</i> message
<code>linkb(9)</code>	link a message block to a <i>STREAMS</i> message
<code>msgdsize(9)</code>	calculate the size of the data in a <i>STREAMS</i> message
<code>msgpullup(9)</code>	pull up bytes in a <i>STREAMS</i> message
<code>pcmsg(9)</code>	test a data block message type for priority control
<code>pullupmsg(9)</code>	pull up the bytes in a <i>STREAMS</i> message
<code>rmvb(9)</code>	remove a message block from a <i>STREAMS</i> message
<code>testb(9)</code>	test if a <i>STREAMS</i> message can be allocated
<code>unbufcall(9)</code>	remove a <i>STREAMS</i> buffer callback
<code>unlinkb(9)</code>	unlink a message block from a <i>STREAMS</i> message

G.1.2 Core UP Queue Functions

<code>backq(9)</code>	find the upstream or downstream queue
<code>bcanput(9)</code>	test flow control on a <i>STREAMS</i> message queue
<code>canenable(9)</code>	test whether a <i>STREAMS</i> message queue can be scheduled
<code>enableok(9)</code>	allow a <i>STREAMS</i> message queue to be scheduled
<code>flushband(9)</code>	flushes band <i>STREAMS</i> messages from a message queue
<code>flushq(9)</code>	flushes messages from a <i>STREAMS</i> message queue
<code>getq(9)</code>	gets a message from a <i>STREAMS</i> message queue

<code>insq(9)</code>	inserts a message into a <i>STREAMS</i> message queue
<code>noenable(9)</code>	disable a <i>STREAMS</i> message queue from being scheduled
<code>OTHERQ(9)</code>	return the other queue of a <i>STREAMS</i> queue pair
<code>putbq(9)</code>	put a message back on a <i>STREAMS</i> message queue
<code>putctl(9)</code>	put a control message on a <i>STREAMS</i> message queue
<code>putctl1(9)</code>	put a 1 byte control message on a <i>STREAMS</i> message queue
<code>putq(9)</code>	put a message on a <i>STREAMS</i> message queue
<code>qenable(9)</code>	schedules a <i>STREAMS</i> message queue service routine
<code>qreply(9)</code>	replies to a message from a <i>STREAMS</i> message queue
<code>qsize(9)</code>	return the number of message on a queue
<code>RD(9)</code>	return the read queue of a <i>STREAMS</i> queue pair
<code>rmvq(9)</code>	remove a message from a <i>STREAMS</i> message queue
<code>SAMESTR(9)</code>	test for <i>STREAMS</i> pipe or <i>FIFO</i>
<code>WR(9)</code>	return the write queue of a <i>STREAMS</i> queue pair

G.1.3 Core MP Queue Functions

<code>canputnext(9)</code>	test flow control on a message queue
<code>canputnext(9)</code>	test flow control on a message queue
<code>freezestr(9)</code>	freeze the state of a stream queue
<code>put(9)</code>	invoke the put procedure for a <i>STREAMS</i> module or driver with a <i>STREAMS</i> message
<code>putnext(9)</code>	put a message on the downstream <i>STREAMS</i> message queue
<code>putnextctl1(9)</code>	put a 1 byte control message on the downstream <i>STREAMS</i> message queue
<code>putnextctl(9)</code>	put a control message on the downstream <i>STREAMS</i> message queue
<code>qprocsoff(9)</code>	disables <i>STREAMS</i> message queue processing for multi-processing
<code>qprocson(9)</code>	enables <i>STREAMS</i> message queue processing for multi-processing
<code>strqget(9)</code>	gets information about a <i>STREAMS</i> message queue
<code>strqset(9)</code>	sets attributes of a <i>STREAMS</i> message queue
<code>unfreezestr(9)</code>	thaw the state of a stream queue

G.1.4 Core DDI/DKI Functions

<code>kmem_alloc(9)</code>	allocate kernel memory
<code>kmem_free(9)</code>	deallocates kernel memory
<code>kmem_zalloc(9)</code>	allocate and zero kernel memory
<code>cmn_err(9)</code>	print a kernel command error
<code>bcopy(9)</code>	copy byte strings
<code>bzero(9)</code>	zero a byte string
<code>copyin(9)</code>	copy user data in from user space to kernel space
<code>copyout(9)</code>	copy user data in from kernel space to user space
<code>delay(9)</code>	postpone the calling process for a number of clock ticks

<code>drv_getparm(9)</code>	driver retrieve kernel parameter
<code>drv_hztomsec(9)</code>	convert kernel tick time between microseconds or milliseconds
<code>drv_htztousec(9)</code>	convert kernel tick time between microseconds or milliseconds
<code>drv_msectohz(9)</code>	convert kernel tick time between microseconds or milliseconds
<code>drv_priv(9)</code>	check if the current process is privileged
<code>drv_usectohz(9)</code>	convert kernel tick time between microseconds or milliseconds
<code>drv_usecwait(9)</code>	delay for a number of microseconds
<code>min(9)</code>	determine the minimum of two integers
<code>max(9)</code>	determine the maximum of two integers
<code>getmajor(9)</code>	get the internal major device number for a device
<code>getminor(9)</code>	get the extended minor device number for a device
<code>makedevice(9)</code>	create a device from a major and minor device numbers
<code>strlog(9)</code>	pass a message to the <i>STREAMS</i> logger
<code>timeout(9)</code>	start a timer
<code>untimeout(9)</code>	stop a timer
<code>mknod(9)</code>	make block or character special files
<code>mount(9)</code>	mount and unmount file systems
<code>umount(9)</code>	mount and unmount file systems
<code>unlink(9)</code>	remove a file

G.1.5 Some Common Extension Functions

<code>linkmsg(9)</code>	link a message block to a <i>STREAMS</i> message
<code>putctl2(9)</code>	put a two byte control message on a <i>STREAMS</i> message queue
<code>putnextctl2(9)</code>	put a two byte control message on the downstream <i>STREAMS</i> message queue
<code>weldq(9)</code>	weld two (or four) queues together
<code>unweldq(9)</code>	unweld two (or four) queues

G.1.6 Some Internal Functions

<code>allocq(9)</code>	allocate a <i>STREAMS</i> queue pair
<code>bcanget(9)</code>	test for message arrival on a band on a stream
<code>canget(9)</code>	test for message arrival on a stream
<code>freeq(9)</code>	deallocate a <i>STREAMS</i> queue pair
<code>qattach(9)</code>	attach a module onto a <i>STREAMS</i> file
<code>qclose(9)</code>	close a <i>STREAMS</i> module or driver
<code>qdetach(9)</code>	detach a module from a <i>STREAMS</i> file
<code>qopen(9)</code>	call a <i>STREAMS</i> module or driver open routine
<code>setq(9)</code>	set sizes and procedures associated with a <i>STREAMS</i> message queue

G.1.7 Some Oddball Functions

<code>appq(9)</code>	append one <i>STREAMS</i> message after another
----------------------	---

<code>esbbscall(9)</code>	install a buffer callback for an extended <i>STREAMS</i> message block
<code>isdatblk(9)</code>	test a <i>STREAMS</i> data block for data type
<code>isdatmsg(9)</code>	test a <i>STREAMS</i> data block for data type
<code>kmem_zalloc_node(9)</code>	allocate and zero memory on a node
<code>msgsize(9)</code>	calculate the size of the message blocks in a <i>STREAMS</i> message
<code>qcountstrm(9)</code>	add all counts on all <i>STREAMS</i> message queues in a stream
<code>xmsgsize(9)</code>	calculate the size of message blocks in a <i>STREAMS</i> message

G.2 Porting from SVR 4.2 MP

This section captures portability information for *SVR 4.2 MP* based systems. If the operating system from which you are porting more closely fits one of the other portability sections, please see that section.

G.2.1 Differences from SVR 4.2 MP

Linux Fast-STREAMS has very few differences from *SVR 4.2 MP*. Not all *SVR 4.2 MP* functions are implemented in the base *Linux Fast-STREAMS* kernel modules. Some functions are included in the *SVR 4.2 MP* compatibility module, ‘`streams-svr4compat.o`’.

G.2.2 Commonalities with SVR 4.2 MP

G.2.3 Compatibility functions for SVR 4.2 MP

<code>itimerout(9)</code>	Perform a timeout at an interrupt level.
<code>lbolt(9)</code>	Time in ticks since reboot.
<code>sleep(9)</code>	Put a process to sleep.
<code>wakeup(9)</code>	Wake a process.
<code>vtop(9)</code>	Convert virtual to physical address.

G.2.3.1 Priority Levels

Linux has a different concept of priority levels than *SVR 4.2 MP*. **Linux** has basically 4 priority levels as follows:

1. Preemptive

At this priority level, software and hardware interrupts are enabled and the kernel is executing with preemption enabled. This means that the currently executing kernel thread could preempt and sleep in favour of another thread of kernel execution.

This priority level only exists on preemptive (mostly 2.6) kernels.

2. Non-Preemptive

At this priority level, software and hardware interrupts are enabled and the kernel is executing with preemption disabled. This means that the currently executing kernel thread will only be interrupted by software or hardware interrupts.

This priority level exists in all kernels.

3. Software Interrupts Disabled

At this priority level, software interrupts are disabled and the kernel is executing with preemption disabled. This means that the currently executing kernel thread will only be interrupted by hardware interrupts.

This is the case when the executing thread is processing a software interrupt, or when the currently executing thread has disabled software interrupts.

This priority level exists in all kernels.

4. Interrupt Service Routines Disabled

At this priority level, hardware interrupts are disabled and the kernel is executing with preemption disabled. This means that the currently executing kernel thread will not be interrupted.

This is the case when the executing thread is processing a hardware interrupt, or when the currently executing thread has disabled hardware interrupts.

This priority level exists in all kernels.

<code>spl0(9)</code>	Set priority level 0.
<code>spl1(9)</code>	Set priority level 1.
<code>spl2(9)</code>	Set priority level 2.
<code>spl3(9)</code>	Set priority level 3.
<code>spl4(9)</code>	Set priority level 4.
<code>spl5(9)</code>	Set priority level 5.
<code>spl7(9)</code>	Set priority level 6.
<code>spl7(9)</code>	Set priority level 7.
<code>spl(9)</code>	Set priority level.
<code>splx(9)</code>	Set priority level x.

G.2.3.2 Atomic Integers

<code>ATOMIC_INT_ADD(9)</code>	Add an integer value to an atomic integer.
<code>ATOMIC_INT_ALLOC(9)</code>	Allocate and initialize an atomic integer.
<code>ATOMIC_INT_DEALLOC(9)</code>	Deallocate an atomic integer.
<code>ATOMIC_INT_DECR(9)</code>	Decrement and test an atomic integer.
<code>ATOMIC_INT_INCR(9)</code>	Increment an atomic integer.
<code>ATOMIC_INT_INIT(9)</code>	Initialize an atomic integer.
<code>ATOMIC_INT_READ(9)</code>	Read an atomic integer.
<code>ATOMIC_INT_SUB(9)</code>	Subtract an integer value from an atomic integer.
<code>ATOMIC_INT_WRITE(9)</code>	Write an integer value to an atomic integer.

G.2.3.3 Basic Locks

LOCK(9)	Lock a basic lock.
LOCK_ALLOC(9)	Allocate a basic lock.
LOCK_DEALLOC(9)	Deallocate a basic lock.
LOCK_OWNED(9)	Determine whether a basic lock is held by the caller.
TRYLOCK(9)	Try to lock a basic lock.
UNLOCK(9)	Unlock a basic lock.

G.2.3.4 STREAMS Locks

MPSTR_QLOCK(9)	Release a queue from exclusive access.
MPSTR_QRELE(9)	Acquire a queue for exclusive access.
MPSTR_STPLOCK(9)	Acquire a stream head for exclusive access.
MPSTR_STPRELE(9)	Release a stream head from exclusive access.

G.2.3.5 Read/Write Locks

RW_ALLOC(9)	Allocate and initialize a read/write lock.
RW_DEALLOC(9)	Deallocate a read/write lock.
RW_RDLOCK(9)	Acquire a read/write lock in read mode.
RW_TRYRDLOCK(9)	Attempt to acquire a read/write lock in read mode.
RW_TRYWRLOCK(9)	Attempt to acquire a read/write lock in write mode.
RW_UNLOCK(9)	Release a read/write lock.
RW_WRLOCK(9)	Acquire a read/write lock in write mode.

G.2.3.6 Sleep Locks

SLEEP_ALLOC(9)	Allocate a sleep lock.
SLEEP_DEALLOC(9)	Deallocate a sleep lock.
SLEEP_LOCK(9)	Acquire a sleep lock.
SLEEP_LOCKAVAIL(9)	Determine whether a sleep lock is available.
SLEEP_LOCKOWNED(9)	Determine whether a sleep lock is held by the caller.
SLEEP_LOCK_SIG(9)	Acquire a sleep lock.
SLEEP_TRYLOCK(9)	Attempt to acquire a sleep lock.
SLEEP_UNLOCK(9)	Release a sleep lock.

G.2.3.7 Synchronization Variables

SV_ALLOC(9)	Allocate a basic condition variable.
SV_BROADCAST(9)	Broadcast a basic condition variable.
SV_DEALLOC(9)	Deallocate a basic condition variable.
SV_SIGNAL(9)	Signal a basic condition variable.
SV_WAIT(9)	Wait on a basic condition variable.
SV_WAIT_SIG(9)	Interruptible wait on a basic condition variable.

G.2.3.8 Resource Allocation

rmalloc(9)	Allocate a number of units from a resource map.
rmap(9)	Allocate a resource map.

<code>rmallocmap_wait(9)</code>	Allocated a resource map.
<code>rmalloc_wait(9)</code>	Allocate a number of units from a resource map.
<code>rmfree(9)</code>	Free a number of units from a resource map.
<code>rmfreemap(9)</code>	Free a resource map.
<code>rmget(9)</code>	Allocated a number of units from a resource map.
<code>rminit(9)</code>	Initialize a resource map.
<code>rmsetwant(9)</code>	Wait for resources on a resource map.
<code>rmwanted(9)</code>	Waiters on a resource map.

G.2.3.9 Device Numbering

<code>major(9)</code>	Get the internal major number of a device.
<code>makedev(9)</code>	Make a device number from internal major and minor device numbers.
<code>minor(9)</code>	Get the internal minor number of a device.

G.2.4 Configuration ala SVR 4.2 MP

G.3 Porting from AIX 5L Version 5.1

G.3.1 Differences from AIX 5L Version 5.1

G.3.2 Commonalities with AIX 5L Version 5.1

G.3.3 Compatibility Functions for AIX 5L Version 5.1

G.3.3.1 Core Extensions

<code>putctl2(9)</code>	Put a 2 byte control message on a <i>STREAMS</i> message queue. <code>putctl2(9)</code> is a <i>Linux Fast-STREAMS</i> core function.
<code>splstr(9)</code>	Set or restore priority levels. <code>splstr(9)</code> is a <i>Linux Fast-STREAMS</i> core function.
<code>splx(9)</code>	Set or restore priority levels. <code>splx(9)</code> is a <i>Linux Fast-STREAMS</i> core function.
<code>weldq(9)</code>	Weld together two pairs of <i>STREAMS</i> message queues. <code>weldq(9)</code> is a <i>Linux Fast-STREAMS</i> core function.
<code>unweldq(9)</code>	Unweld two pairs of <i>STREAMS</i> message queues. <code>unweldq(9)</code> is a <i>Linux Fast-STREAMS</i> core function.

G.3.3.2 Common Module Utilities

<code>mi_bufcall(9)</code>	Reliable alternative to <code>bufcall(9)</code> .
<code>mi_close_comm(9)</code>	<i>STREAMS</i> common minor device close utility.
<code>mi_next_ptr(9)</code>	<i>STREAMS</i> minor device list traversal.
<code>mi_open_comm(9)</code>	<i>STREAMS</i> common minor device open utility.
<code>mi_prev_ptr(9)</code>	<i>STREAMS</i> minor device list traversal.

G.3.3.3 Registration

`str_install(9)` Install a *STREAMS* module or driver.

G.3.3.4 Message Filtering

`wantio(9)` Perform direct I/O from a *STREAMS* driver.

`wantmsg(9)` Provide a filter of wanted messages from a *STREAMS* module.

G.3.4 Configuration ala AIX 5L Version 5.1

G.4 Porting from HP-UX 11.0i v2

G.4.1 Differences from HP-UX 11.0i v2

G.4.2 Commonalities with HP-UX 11.0i v2

G.4.3 Compatibility Functions for HP-UX 11.0i v2

G.4.3.1 Core Extensions

`streams_put(9)` Invoke the put procedure for a *STREAMS* module or driver with a *STREAMS* message. `streams_put(9)` is implemented using `put(9)`. `put(9)` is a *Linux Fast-STREAMS* core function.

`putctl2(9)` Put a 2 byte control message on a *STREAMS* message queue. `putctl2(9)` is a *Linux Fast-STREAMS* core function.

`putnextctl2(9)` Put a 2 byte control message on the downstream *STREAMS* message queue. `putnextctl2(9)` is a *Linux Fast-STREAMS* core function.

`unweldq(9)` Unweld two pairs of streams queues. `unweldq(9)` is a *Linux Fast-STREAMS* core function.

`weldq(9)` Weld together two pairs of streams queues. `weldq(9)` is a *Linux Fast-STREAMS* core function.

G.4.3.2 Registration

`str_install(9)` Install a *STREAMS* module or driver.

`str_uninstall(9)` Uninstall a *STREAMS* module or driver.

G.4.3.3 Sleeping

`streams_get_sleep_lock(9)` Provide access to the global sleep lock.

G.4.4 Configuration ala HP-UX 11.0i v2

G.5 Porting from OSF/1 1.2/Digital UNIX

G.5.1 Differences from OSF/1 1.2/Digital UNIX

G.5.2 Commonalities with OSF/1 1.2/Digital UNIX

G.5.3 Compatibility Functions for OSF/1 1.2/Digital UNIX

G.5.3.1 Core Extensions

<code>lbolt(9)</code>	Time in ticks since reboot <code>lbolt(9)</code> is a <i>Linux Fast-STREAMS</i> core function.
<code>puthere(9)</code>	Invoke the put procedure for a <i>STREAMS</i> module or driver with a <i>STREAMS</i> message. <code>puthere(9)</code> is implemented using <code>put(9)</code> . <code>put(9)</code> is a <i>Linux Fast-STREAMS</i> core function.
<code>weldq(9)</code>	Weld together two pairs of streams queues. <code>weldq(9)</code> is a <i>Linux Fast-STREAMS</i> core function.
<code>unweldq(9)</code>	Unweld two pairs of streams queues. <code>unweldq(9)</code> is a <i>Linux Fast-STREAMS</i> core function.

G.5.3.2 Common Module Utilities

<code>streams_close_comm(9)</code>	Common minor device close utility.
<code>streams_open_comm(9)</code>	Common minor device open utility.
<code>streams_open_ocomm(9)</code>	Common minor device open utility.

G.5.3.3 Registration

<code>strmod_add(9)</code>	Add a <i>STREAMS</i> module.
<code>strmod_del(9)</code>	Delete a <i>STREAMS</i> module or driver from the kernel.

G.5.3.4 Others

<code>time(9)</code>	(undoc).
----------------------	----------

G.5.4 Configuration ala OSF/1 1.2/Digital UNIX

G.6 Porting from UnixWare 7.1.3 (OpenUnix 8)

G.6.1 Differences from UnixWare 7.1.3 (OpenUnix 8)

G.6.2 Commonalities with UnixWare 7.1.3 (OpenUnix 8)

UnixWare provides most of the core functions provide by *Linux Fast-STREAMS* along with all of the compatibility functions provided by the *SVR 4.2 MP* compatibility module. In addition the functions provided here in the *UnixWare* compatibility module are provided.

G.6.3 Compatibility Functions for UnixWare 7.1.3 (OpenUnix 8)

The following compatibility functions are in addition to all *SVR 4.2* compatibility functions.

G.6.3.1 Device Numbering

Device numbering has evolved since *UNIX System V Release 3.0* and provides internal, external and extended device numbering. These functions are provided for backward compatibility with some drivers that were written for the older system. These are core functions in the *Linux Fast-STREAMS* implementation.

<code>emajor(9)</code>	Get the external (real) major device number from the device number.
<code>eminor(9)</code>	Get the external extended minor device number from the device number.
<code>etoimajor(9)</code>	Convert an external major device number to an internal major device number.
<code>getemajor(9)</code>	Get the external (real) major device number.
<code>geteminor(9)</code>	Get the external minor device number.
<code>itoemajor(9)</code>	Convert an internal major device number to an external major device number.

G.6.3.2 Memory Alignment

In attempting to unify several disparaging *UNIX*-based systems (in particular *XENIX* and *UnixWare*, it became necessary to sometimes address the alignment of data buffers. Certainly a better way to accomplish this would be to allocate data buffers using other allocators that provide the required alignment and other buffer characteristics and then allocating a message and data block with a call to `esballoc(9)`. Nevertheless, these functions were provided for making message blocks, data blocks and data buffers meet specific physical requirements.

Linux Fast-STREAMS provides these functions for compatibility, however, most of the physical requirements provided are ignored.

<code>allocb_physreq(9)</code>	Allocate a <i>STREAMS</i> message and data block.
<code>msgphysreq(9)</code>	Cause a message block to meet physical requirements.
<code>msgpullup_physreq(9)</code>	Pull up bytes in a <i>STREAMS</i> message.
<code>msgscgth(9)</code>	(undoc).

G.6.3.3 Direct *STREAMS* Input-Output Controls

<code>striocall(9)</code>	(undoc).
---------------------------	----------

G.6.4 Configuration ala UnixWare 7.1.3 (OpenUnix 8)

G.7 Porting from Solaris 9/SunOS 5.9

G.7.1 Differences from Solaris 9/SunOS 5.9

G.7.2 Commonalities with Solaris 9/SunOS 5.9

G.7.3 Compatibility Functions for Solaris 9/SunOS 5.9

G.7.3.1 STREAMS Queue Referenced Callbacks

<code>qbufcall(9)</code>	Install a <i>STREAMS</i> buffer callback.
<code>qunbufcall(9)</code>	Cancel a <i>STREAMS</i> buffer callback.
<code>qtimeout(9)</code>	Start a timer associated with a queue.
<code>quntimeout(9)</code>	Stop a timer associated with a queue.
<code>qwait(9)</code>	Wait for a queue message.
<code>qwait_sig(9)</code>	Wait for a queue message or signal.
<code>queclass(9)</code>	Return the class of a <i>STREAMS</i> message.
<code>qwriter(9)</code>	<i>STREAMS</i> mutex upgrade.

G.7.3.2 STREAMS Registration

<code>install_driver(9)</code>	Install a device driver.
<code>mod_info(9)</code>	Provides information on a loadable kernel module to the <i>STREAMS</i> executive.
<code>mod_install(9)</code>	Installs a loadable kernel module in the <i>STREAMS</i> executive.
<code>mod_remove(9)</code>	Removes a loadable module from the <i>STREAMS</i> executive.

G.7.3.3 DDI

Solaris provides a wide array of Device Driver Interface functions available for use by device drivers. Many of these functions are useful for *STREAMS* device and pseudo-device drivers and modules. Almost all of these functions, however, are *Solaris*-specific and are completely non-portable to other *UNIX*-based operating systems. To make matters worse for portability, many of these functions have no *SVR 4.2 MP* equivalents.

<code>ddi_create_minor_node(9)</code>	Create a minor node for this device.
<code>ddi_remove_minor_node(9)</code>	Remove a minor node for a device.
<code>ddi_driver_major(9)</code>	Find the major device number associated with a driver.
<code>ddi_getiminor(9)</code>	Get the internal minor device number.
<code>ddi_driver_name(9)</code>	Return normalized driver name.
<code>ddi_get_cred(9)</code>	Get a reference to the credentials of the current user.
<code>ddi_get_instance(9)</code>	Get device instance number.
<code>ddi_get_lbolt(9)</code>	Get the current value of the system tick clock.
<code>ddi_get_pid(9)</code>	Get the process id of the current process.
<code>ddi_get_time(9)</code>	Get the current time in seconds since the epoch.
<code>ddi_removing_power(9)</code>	
<code>ddi_get_soft_state(9)</code>	
<code>ddi_soft_state(9)</code>	
<code>ddi_soft_state_fini(9)</code>	
<code>ddi_soft_state_free(9)</code>	
<code>ddi_soft_state_init(9)</code>	
<code>ddi_soft_state_zalloc(9)</code>	
<code>ddi_umem_alloc(9)</code>	Allocate page aligned kernel memory.
<code>ddi_umem_free(9)</code>	Free page aligned kernel memory.

G.7.3.4 Loadable Module Interface

<code>_fini(9)</code>	
<code>_info(9)</code>	
<code>_init(9)</code>	
<code>attach(9)</code>	Attach a device to the system or resume a suspended device.
<code>getinfo(9)</code>	
<code>identify(9)</code>	Determine if a driver is associated with a device.
<code>detach(9)</code>	Detach a device from the system or suspend a device.
<code>power(9)</code>	Power a device attached to the system.
<code>probe(9)</code>	

G.7.4 Configuration ala Solaris 9/SunOS 5.9

G.8 Porting from Super/UX

G.8.1 Differences from Super/UX

G.8.2 Commonalities with Super/UX

G.8.3 Compatibility Functions for Super/UX

<code>lbolt(9)</code>	time in ticks since reboot
-----------------------	----------------------------

G.8.4 Configuration ala Super/UX

G.9 Porting from UXP/V

G.9.1 Differences from UXP/V

G.9.2 Commonalities with UXP/V

G.9.3 Compatibility Functions for UXP/V

G.9.4 Configuration ala UXP/V

G.10 Porting from LiS 2.18.1

G.10.1 Differences from LiS 2.18.1

G.10.2 Commonalities with LiS 2.18.1

G.10.3 Compatibility Functions for LiS 2.18.1

G.10.3.1 Extensions

<code>lis_appq(9)</code>	Append one <i>STREAMS</i> message after another.
--------------------------	--

<code>lis_date(9)</code>	
<code>lis_esbbcall(9)</code>	Install a buffer callback for an extended <i>STREAMS</i> message block.
<code>lis_find_strdev(9)</code>	
<code>lis_OTHER(9)</code>	Return the other queue of a <i>STREAMS</i> queue pair.. This function is intended to accommodate a common miss-spelling of <code>OTHERQ(9)</code> .
<code>lis_version(9)</code>	
<code>lis_xmsgsize(9)</code>	Calculate the size of message blocks in a <i>STREAMS</i> message.

G.10.3.2 Device Creation and Deletion

<code>lis_mknod(9)</code>	Make block or character special files.
<code>lis_unlink(9)</code>	Remove a file.
<code>lis_mount(9)</code>	Mount a file system.
<code>lis_umount2(9)</code>	Unmount a file system.
<code>lis_umount(9)</code>	Unmount a file system.

G.10.3.3 Registration

<code>lis_register_strdev(9)</code>	Register a <i>STREAMS</i> device.
<code>lis_register_strmod(9)</code>	Register a <i>STREAMS</i> module.
<code>lis_unregister_strdev(9)</code>	Unregister a <i>STREAMS</i> device.
<code>lis_unregister_strmod(9)</code>	Unregister a <i>STREAMS</i> module.

G.10.4 Configuration ala LiS 2.18.1

G.11 Developing Portable STREAMS Modules

In the process of creating the *Linux Fast-STREAMS* subsystem in such a way so as to facilitate portability of *STREAMS* drivers and modules from a wide range of *UNIX* operating system variants, a number of guidelines for the development of portable *STREAMS* drivers and modules have been developed. These guidelines, when adhered to, will allow the resulting driver or module to be ported to another *STREAMS* implementation with minimal effort. These portability guidelines are collected here.

G.11.1 Memory Allocation

Portable *STREAMS* modules and drivers will always allocate memory using the *SVR4* memory allocators/deallocators: `kmem_alloc(9)`, `kmem_zalloc(9)` and `kmem_free(9)`.

Additional eligible allocators are:

```
rmallocmap(9)  rmfreemap(9)  rmalloc(9)  rmalloc_wait(9)  rmfree(9)  rminit(9)
rmsetwant(9)  rmwanted(9)
```

Unfortunately, these resource map allocators are not available on *AIX* so, if portability to the *AIX* is important, then do not use these allocators.

Additional eligible allocators are:

`kmem_fast_alloc(9)` `kmem_fast_free(9)`

G.11.2 Alignment of Message Buffers

G.11.3 Disabling and Enabling Queue Procedures

Portable *STREAMS* modules and drivers will always call `qprocson(9)` before returning from its queue open procedure (see `qopen(9)`).

Portable *STREAMS* modules and drivers will always call `qprocsoff(9)` upon entering its queue close procedure (see `qclose(9)`).

G.11.4 Freezing and Unfreezing Streams

G.11.5 Passing Messages from Interrupt Service Routines

G.11.6 Timeout Call Back and Link Identifiers

Although buffer callbacks identifiers (see `bufcall(9)`), timeout identifiers (see `timeout(9)`), and multiplexing driver link identifiers (see `I_LINK` and `I_PLINK` under `streamio(7)`), are often illustrated as small integer numbers, with some *STREAMS* implementations, including *Linux Fast-STREAMS*, these identifiers are kernel addresses (pointers) and are never small integer values like 1, 2, or 3.

Also, there is no guarantee that the identifier will be positive. It is guaranteed that the returned identifier will not be zero (0). Zero is used by these function as a return value to indicate an error.

Portable *STREAMS* drivers and modules will not depend upon the returned identifier from `bufcall(9)`, `timeout(9)` or `streamio(7)` as being any specific range of value. Portable drivers and modules will save any returned identifiers in data types that will not lose the precision of the identifier.

G.11.7 Synchronization with Timeouts and Callback Functions

G.11.8 Synchronization with Callout Functions

G.11.9 Synchronization of Drivers and Modules

G.11.10 Special *STREAMS* Message Types

G.11.11 Use of Message Allocation Priorities

G.11.12 Registration and Deregistration

G.11.13 Device Numbering

UNIX Device Numbering

In versions of *UNIX System V* previous to *Release 4*, the major and minor device numbers were each 8 bit, and they were packed into a 16 bit word (usually a C Language *short* variable). Under *UNIX System V Release 4*, the device numbers are held in a `dev_t` variable, which is often implemented as a 32 bit integer. The minor device number is held as 14 bits, and a further 8 bits are used for the major device number. `dev_t` is often referred to as the "expanded device type", since it allows many more minor devices than before.

Many drivers were written for earlier releases, and may eventually be ported to *UNIX System V Release 4*. In earlier releases, some manufacturers got around the 256 minor device number limit by using multiple major device numbers for a device. Devices were created with different major device numbers (the external major device number) but they all mapped to the same device driver entry in the device switch tables (the internal device number). Even under this scheme, each major device could only support 256 minor devices, but the driver could support many more. This has been recognized in *UNIX System V Release 4*, and functions are provided to do this mapping; for example, the function `etoimajor()` and so on, give a machine independent interface to the device number mapping.¹

Linux Device Numbering

Versions of the **Linux** kernel in the 2.4 kernel series and prior to 2.6 also provided an 8 bit major device number and an 8 bit minor device number grouped into a 16-bit combined device number. Linux 2.6 kernels (and some patched 2.4 kernels) now have larger device numbers. These extended device numbers are 12 bits for major device number and 20 bits for minor device number, with 32 bits for the combined device number.

LiS Device Numbering

LiS prior to the 2.18.0 release was incapable of providing an internal representation of the device number and the number of minor device numbers for a device driver was restricted to 256, just as in *UNIX System V Release 3.2*.² Many **OpenSS7** device drivers written for *LiS* would allocate additional major device numbers if required. Good examples of devices that require more than 255 minor device numbers are *INET* clone devices, *SCTP* streams, *SS7* signalling link streams, *MG* media channels, etc. These streams are often *I_PLINKed* under a multiplexing driver and do not even consume a system file descriptor.

Linux Fast-STREAMS Device Numbering

Linux Fast-STREAMS began with extended device numbering. The ‘*specfs*’ shadow special character device file system used by *Linux Fast-STREAMS* uses the ‘*inode*’ number to hold the `dev_t` device number instead of the ‘*inode->i_rdev*’, which on older kernels is only a 16-bit *short*.

¹ *The Magic Garden Explained*

² Actually, 255 as the kernel reserved minor device number 255 for expansion.

In earlier versions of *Linux Fast-STREAMS*, the internal device numbering is 16-bits for major device number and 16-bits for minor device number. This will soon be changed to 12-bits for major device number and 20-bits for minor device number to accommodate the newer **Linux** scheme.

On 2.6 **Linux** kernels that support the newer extended device numbers, external device numbers and internal device numbers will be the same. On 2.4 **Linux** kernels with the older 16-bit device numbers, external device number and internal device numbers will differ. In some situations, an internal device number can exist with no corresponding external device number (accessed only via a clone device or direct access to the mounted ‘**specfs**’ shadow special character device file system).

<code>etoimajor(9)</code>	change external to internal major device number
<code>getemajor(9)</code>	get external major device number
<code>geteminor(9)</code>	get external minor device number
<code>itoemajor(9)</code>	change internal to external major device number

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.

Index

- (
 (none) 136
- /
- /dev 150
 /etc/master.d 150
 /etc/strload.conf 267, 268
 /proc file system 273
-
- _AIX_SOURCE 224
 _fini(9) 241, 342
 _HPUX_SOURCE 224
 _info(9) 241, 342
 _init(9) 241, 342
 _LFS_SOURCE 224
 _LIS_SOURCE 224
 _OSF_SOURCE 224
 _SUN_SOURCE 224
 _SVR4_SOURCE 224
 _UW7_SOURCE 224
- 0**
- 0 108
- A**
- adjmsg(9) 21, 229, 296, 331
 administration, STREAMS 255
 administrative driver, STREAMS 35
 administrative utilities 255
 AIX 10, 25, 236, 237, 247, 254, 343
 AIX 5L Version 5.1 236, 337
 AIX 5L Version 5.1, commonalities 337
 AIX 5L Version 5.1, compatibility 329
 AIX 5L Version 5.1, compatibility functions .. 236, 337
 AIX 5L Version 5.1, configuration 338
 AIX 5L Version 5.1, differences 337
 AIX 5L Version 5.1, portability 337
 AIX PSE 236, 329, 337, 338
 AIX, compatibility module 237
 AIX, core functions 236
 allocb(9) ... 21, 22, 31, 54, 59, 88, 91, 92, 134, 229, 271, 273, 296, 297, 299, 303, 307, 331
 allocb_physreq(9) 241, 340
 allocq(9) 232, 333
 ANYMARK 67
 apinfo(9) 228
- application interface, *STREAMS* 31
 appq(9) 233, 333
 ARPANET 4
 ASCEBC 51
 asynchronous callbacks 254
 asynchronous callouts 254
 asynchronous entry points 254
 asynchronous stream example 50
 ATOMIC_INT_ADD(9) 234, 335
 ATOMIC_INT_ALLOC(9) 234, 335
 ATOMIC_INT_DEALLOC(9) 234, 335
 ATOMIC_INT_DECR(9) 234, 335
 ATOMIC_INT_INCR(9) 234, 335
 ATOMIC_INT_INIT(9) 234, 335
 ATOMIC_INT_READ(9) 234, 335
 ATOMIC_INT_SUB(9) 234, 335
 ATOMIC_INT_WRITE(9) 234, 335
 attach(9) 241, 342
 autopush(8) 35
 autopush(8) 247, 255
 autopush_add(9) 35, 229
 autopush_del(9) 35, 229
 autopush_find(9) 35, 229
 autopush_vml(9) 35, 229
- B**
- b_band 296, 304
 b_next 301
 b_rptr 297, 299
 b_wptr 297, 299
 B300 217
 backq(9) 23, 230, 296, 331
 bandinfo(9) 228
 bcanget(9) 233, 333
 bcanput(9) .. 23, 24, 66, 73, 133, 134, 230, 296, 331
 bcanputnext(9) 24, 230
 bcopy(9) 231, 332
 bcopy(from, to, nbytes) 136
 benefits of *STREAMS* 6
 BIND_REQ 86, 88
 BOUND 86, 88
 BPRI_HI 22, 296
 BPRI_LO 22, 296
 BPRI_MED 22, 296
 BRPI_WAITOK 22
 BSD 11
 bufcall(9) 21, 22, 74, 88, 89, 91, 92, 134, 227, 229, 251, 253, 297, 307, 331, 344
 buffcall(9) 237, 337
 BUSY 156
 bzero(9) 231, 332

bzero(buffer, nbytes) 136

C

canenable(9) 230, 331
 canget(9) 233, 333
 CANONPROC 51
 CANONPROC 53, 55, 56
 canput(9) ... 23, 24, 73, 74, 75, 133, 134, 141, 144,
 145, 160, 230, 297, 301
 canputnext(9) 24, 230, 332
 CBAUD 214, 216
 cdevsw(9) 26, 227
 cdevsw_list(9) 26
 CE_CONT 309
 CE_IPANIC 309
 CE_NOTE 309
 CE_PANIC 309
 CE_WARN 309
 CHARPROC 51, 53, 54, 55, 56
 clone device 248
 clone(4) 225, 248, 249
 clone, driver 225
 Clone, driver 225
 CLONEOPEN 108, 131, 161, 164, 184
 close(2) 31, 37, 61, 226
 cmn_err(9) 232, 309, 332
 cmn_err(level,...) 136
 cmpdev(x) 137
 common extension functions 333
 commonalities, AIX 5L Version 5.1 337
 commonalities, HP-UX 11.0i v2 338
 commonalities, LiS 2.18.1 342
 commonalities, OSF/1 1.2/Digital UNIX 339
 commonalities, Solaris 9/SunOS 5.9 340
 commonalities, Super/UX 342
 commonalities, SVR 4.2 MP 334
 commonalities, UnixWare 7.1.3 (OpenUnix 8)
 339
 commonalities, UXP/V 342
 compatibility 233
 compatibility functions, AIX 5L Version 5.1 .. 236,
 337
 compatibility functions, HP-UX 11.0i v2.. 238, 338
 compatibility functions, LiS 2.18.1 242, 342
 compatibility functions, OSF/1 1.2/Digital UNIX
 240, 339
 compatibility functions, Solaris 9/SunOS 5.9
 241, 340
 compatibility functions, Super/UX 342
 compatibility functions, SVR 4.2 MP 233, 334
 compatibility functions, UnixWare 7.1.3
 (OpenUnix 8) 240, 339
 compatibility functions, UXP/V 342
 compatibility module, AIX 237
 compatibility module, HP-UX 239

compatibility module, OSF/1 240
 compatibility module, Solaris 241
 compatibility module, SVR 4.2 MP 233
 compatibility module, UnixWare 241
 compatibility, AIX 5L Version 5.1 329
 compatibility, HP-UX 11.0i v2 329
 compatibility, LiS 2.18.1 329
 compatibility, OSF/1 1.2/Digital 329
 compatibility, Solaris 9/SunOS 5.9 329
 compatibility, Super/UX 329
 compatibility, SVR 4.2 MP 329
 compatibility, UnixWare 7.1.3 (OpenUnix 8) .. 329
 compatibility, UXP/V 329
 compliance, UNIX 03 12
 compliance, UNIX 95 13
 compliance, UNIX 98 13
 concepts 16
 configuration 251
 configuration, AIX 5L Version 5.1 338
 configuration, HP-UX 11.0i v2 338
 configuration, LiS 2.18.1 343
 configuration, OSF/1 1.2/Digital UNIX 339
 configuration, Solaris 9/SunOS 5.9 342
 configuration, STREAMS 251
 configuration, Super/UX 342
 configuration, SVR 4.2 MP 337
 configuration, UnixWare 7.1.3 (OpenUnix 8) .. 340
 configuration, UXP/V 342
 conformance 329
 connld(4) 225
 connld, module 225
 contributors 1
 copyb(9) 21, 229, 297, 298, 331
 copyin(9) 109, 113, 231, 332
 copymsg(9) 21, 229, 298, 331
 copyout(9) 109, 113, 231, 332
 core ddi/dki functions 332
 core functions, AIX 236
 core functions, HP-UX 238
 core functions, OSF/1 240
 core functions, Solaris 241
 core functions, SVR 4.2 MP 233
 core functions, UnixWare 240
 core message functions 331
 core queue functions, MP 332
 core queue functions, UP 331
 crash(1M) 309, 310
 CREAD 217
 cred.h 282
 cred_t(9) 228
 credits 1
 Criticism of STREAMS 7
 CS8 217
 CSWIDTH 211, 212

D

D_MP 253
D_MTAPAIR 252
D_MTPERM0D 252
D_MTPERQ 253
 Data Block 31
 Data Block Types 31
 Data Buffer 31
datab 300
datab(9) 88
datab(9) 228
datamsg(9) 21, 129, 229, 298, 307, 331
db_base 299
db_freep 299
db_lim 299
dbinfo(9) 228
dbl_k_t 298
DDI 13
ddi.h 135
DDI/DKI 247, 251, 332, 341
ddi_create_minor_node(9) 241, 341
ddi_driver_major(9) 241, 341
ddi_driver_name(9) 241, 341
ddi_get_cred(9) 241, 341
ddi_get_instance(9) 241, 341
ddi_get_lbolt(9) 241, 341
ddi_get_pid(9) 242, 341
ddi_get_soft_state(9) 242, 341
ddi_get_time(9) 242, 341
ddi_getimino(9) 241, 341
ddi_remove_minor_node(9) 242, 341
ddi_removing_power(9) 242, 341
ddi_soft_state(9) 242, 341
ddi_soft_state_fini(9) 242, 341
ddi_soft_state_free(9) 242, 341
ddi_soft_state_init(9) 242, 341
ddi_soft_state_zalloc(9) 242, 341
ddi_umem_alloc(9) 242, 341
ddi_umem_free(9) 242, 341
 definitions, *STREAMS* 15
delay(9) 231, 332
delay(ticks) 136
DELETE 47
 design guidelines 131
detach(9) 242, 342
dev_t 345
 developing portable streams modules 343
 device numbers 247
 device numbers, external 247
 device numbers, internal 247
devinfo(9) 227
 differences, AIX 5L Version 5.1 337
 differences, HP-UX 11.0i v2 338
 differences, LiS 2.18.1 342
 differences, OSF/1 1.2/Digital UNIX 339

differences, Solaris 9/SunOS 5.9 340
 differences, Super/UX 342
 differences, SVR 4.2 MP 334
 differences, UnixWare 7.1.3 (OpenUnix 8) 339
 differences, UXP/V 342
 Digital UNIX 10
 Does Linux have STREAMS? 3
 driver example 245
 Driver Switch Table 25
 driver-kernel interface 128
 drivers, clone 225
 drivers, Clone 225
 drivers, echo 225
 drivers, Echo 225
 drivers, fifo 225
 drivers, FIFO 225
 drivers, log 225
 drivers, Log 225
 drivers, Named STREAMS Device 225
 drivers, nsdev 225
 drivers, Null STREAM 225
 drivers, nuls 225
 drivers, pipe 225
 drivers, Pipe 225
 drivers, sad 225
 drivers, STREAMS 225
 drivers, STREAMS Administrative Driver 225
drv_getparm(9) 130, 132, 232, 333
drv_hztousec(9) 333
drv_hzto msec(9) 232, 333
drv_hztousec(9) 232
drv_msectohz(9) 232, 333
drv_priv(9) 108, 232, 333
drv_setparm(9) 130, 132
drv_usec to hz(9) 232, 333
drv_usecwait(9) 232, 333
dupb(9) 21, 229, 298, 300, 331
dupmsg(9) 21, 60, 131, 229, 298, 331

E

EAGAIN 197, 220, 287
EBADF 200
EBADMSG 64
EBUSY 157
EBUSY 201
echo(4) 225
 echo, driver 225
 Echo, driver 225
ed(1) 321
EFAULT 113, 124
EINTR 196
EINVAL 63, 110, 124, 180, 193, 200, 220
EIO 84, 103
emajor(9) 241, 340
EMFILE 196

Index

`emisor(9)` 241, 340
`enableok(9)` 23, 74, 230, 299, 331
`ENFILE` 196
`ENOMEM` 196
`ENOSR` 196
`ENXIO` 198, 220, 291
`EPERM` 66, 306
`EPROTO` 290
`EREMOTE` 201
`errno(3)` 82, 84, 113, 283
`errno.h` 135
`ERROR_ACK` 82, 86, 88
`esballo(9)` .. 21, 22, 31, 93, 94, 229, 251, 299, 300, 331, 340
`esbcall(9)` 22, 233, 334
`esbbufcall(9)` 251
`etoimajor(9)` 241, 248, 340, 346
`euc.h` 211, 212
`euioctl.h` 211
`examples` 245
`exit(2)` 100
`expdev(x)` 137
`extended buffers` 92
`extensions` 232
`extensions from LiS 2.18.1` 233
`external device numbers` 247

F

`FALSE` 160, 298
`FASTBUF` 22
`fattach(2)` 31, 225
`fattach(8)` 255
`fattach(8)` 257
`fcntl(2)` 31, 226
`fdetach(2)` 31, 225
`fdetach(8)` 255
`fdetach(8)` 258
`fifo(4)` 225
`fifo, driver` 225
`FIFO, driver` 225
`FIFOs, STREAMS` 36
`file.h` 135
`files` 223
`flush handling` 124
`FLUSHALL` 143, 299
`FLUSHBAND` 124, 291
`flushband(9)` 23, 66, 127, 230, 299, 331
`FLUSHDATA` 143, 299
`flushq(9)` 23, 230, 299, 331
`FLUSHR` 124, 125, 127, 157, 198, 218, 220, 291
`FLUSHRW` 124, 127, 218, 290
`FLUSHW` 124, 125, 126, 127, 143, 157, 198, 218, 220, 290, 291
`FMNAMESZ` 124
`fmodsw(9)` 26, 27, 227

`fmodsw_list(9)` 26
`free_rtn` 299
`freeb(9)` 21, 89, 93, 134, 229, 300, 331
`FreeBSD` 4
`freemsg(9)` 21, 89, 90, 227, 229, 299, 300, 331
`freeq(9)` 233, 333
`freezestr(9)` 230, 254, 332
`fstat(2)` 250
`FULL` 160

G

`GCOM` 10
`GETADDR` 116
`getadmin(9)` 23, 229, 300
`getemajor(9)` 241, 247, 248, 340, 346
`getemajor(x)` 136
`getemisor(9)` 241, 247, 248, 340, 346
`getemisor(x)` 137
`GETINDATA` 123
`getinfo(9)` 242, 342
`getmajor(9)` 231, 248, 249, 333
`getmajor(x)` 136
`getmid(9)` 23, 229, 300
`getminor(9)` 231, 248, 333
`getminor(x)` 137
`getmsg(2)` 31, 37, 54, 55, 61, 75, 76, 81, 82, 83, 84, 225, 226, 281, 285
`getpgid(2)` 100
`getpgrp(2)` 100
`getpmsg(2)` 31, 62, 225, 226
`getq(9)` .. 23, 65, 71, 73, 74, 75, 133, 134, 141, 145, 160, 230, 300, 303, 331
`getsid(2)` 100
`GETSTRUCT` 116, 123
`grep(1)` 321

H

`HP-UX` 10, 25, 238, 239, 240, 247, 329, 338
`HP-UX 11.0i v2` 238, 338
`HP-UX 11.0i v2, commonalities` 338
`HP-UX 11.0i v2, compatibility` 329
`HP-UX 11.0i v2, compatibility functions` .. 238, 338
`HP-UX 11.0i v2, configuration` 338
`HP-UX 11.0i v2, differences` 338
`HP-UX 11.0i v2, portability` 338
`HP-UX 11.0i v2, STREAMS/UX` 238
`HP-UX, compatibility module` 239
`HP-UX, core functions` 238
`HUPCL` 217
`HZ` 92, 137

I

I_ANCHOR 227
I_ATMARK 29, 66, 67, 226
I_AUTOPUSH 227
I_BIGPIPE 227
I_CANPUT 29, 66, 67, 227
I_CKBAND 29, 66, 226
I_CLTIME 271
I_E_RECVFD 226
I_FATTACH 227
I_FDETACH 227
I_FDINSERT 29, 226
I_FIFO 227
I_FIND 29, 226
I_FLUSH 29, 226
I_FLUSHBAND 29, 66, 226
I_GERROPT 227
I_GETBAND 29, 66, 67, 226
I_GETCLTIME 29, 227
I_GETPMSG 227
I_GETSIG 29, 226
I_GETTP 227
I_GRDOPT 29, 226
I_GWROPT 29, 197, 226, 273
I_HEAP_REPORT 227
I_LINK ... 5, 9, 17, 27, 29, 173, 174, 175, 178, 179,
180, 184, 185, 186, 187, 190, 193, 226, 282, 344
I_LIST 29, 123, 124, 226
I_LOOK 29, 226
I_NREAD 226
I_PEEK 29, 226
I_PIPE 227
I_PLINK ... 9, 17, 27, 29, 178, 191, 192, 193, 226,
282, 344, 345
I_POP 17, 29, 44, 48, 105, 109, 139, 226
I_PUNLINK ... 9, 17, 29, 191, 192, 193, 226, 282
I_PUSH ... 17, 29, 44, 46, 51, 105, 108, 139, 175,
196, 226, 272
I_PUTPMSG 227
I_RECVFD 29
I_RECVFD 201
I_RECVFD 203, 226, 284
I_S_RECVFD 227
I_SENDFD 29
I_SENDFD 201
I_SENDFD 226, 284
I_SERROPT 227
I_SETCLTIME 29, 227
I_SETSIG 29, 95, 98, 99, 226, 288
I_SRDOPT 29, 63, 226
I_STATS 227
I_STR ... 8, 29, 47, 48, 110, 111, 112, 114, 116, 118,
132, 161, 165, 211, 226, 279, 282, 283, 284, 291
I_SWROPT 29, 197, 226, 273

I_UNLINK .. 9, 17, 29, 177, 178, 180, 182, 185, 186,
187, 190, 191, 193, 226, 282
iBCS 5, 36
ICANON 207, 209
identify(9) 242, 342
IDLE 86, 88
IEXTEN 209
inet(4) 249
INFPSZ 107
input and output controls 109
input and output polling 95
input-output controls, STREAMS 226
insf(8) 35
insf(8) 247, 255
insf(8) 259
insq(9) 23, 69, 74, 145, 230, 301, 302, 332
install_driver(9) 242, 341
Intel Binary Compatibility Suite 5, 36
Intel Binary Compatibility Suite (iBCS) 12
internal device numbers 247
internal functions 333
introduction 3
INUSE 217
ioctl(2) ... 17, 24, 29, 31, 37, 57, 95, 98, 99, 109,
161, 196
ioctl(2) 201
ioctl(2) 226, 282
IOCWAIT 30
isastream(2) 31, 225
isdatblk(9) 233, 334
isdatamsg(9) 233, 334
ISPTM 220
itimeout(9) 233, 334
itoemajor(9) 241, 248, 340, 346
IXANY 213
IXOFF 213
IXON 209, 213

J

JWINSIZE 209, 214, 216

K

kernel level facilities 32
kernel level facilities, STREAMS 32
kill(2) 100, 105
kmem_alloc(9) 231, 273, 332
kmem_alloc(9) 343
kmem_fast_alloc(9) 231, 344
kmem_fast_free(9) 231, 344
kmem_free(9) 231, 332
kmem_free(9) 343
kmem_zalloc(9) 231, 273, 332
kmem_zalloc(9) 343

kmem_zalloc_node(9) 233, 334

L

l_index 179, 180, 186
 l_qbot 180, 186, 187
 l_qtop 186
 LASTMARK 67
 lbolt 137
 lbolt(9) 233, 240, 334, 339, 342
 license, FDL 353
 license, GNU Free Documentation License 353
 license, GNU General Public License 347
 license, GPL 347
 linkb(9) 21, 90, 229, 301, 331
 linkmsg(9) 232, 333
 Linux 3, 233, 234, 235
 Linux Fast-STREAMS 25
 Linux Fast-STREAMS (LfS) 4, 12
 Linux STREAMS (LiS) 3, 10, 233, 242, 247, 329,
 342, 343, 345
 LiS 25, 233, 242, 342, 343
 LiS 2.18.1 233, 242, 342, 343
 LiS 2.18.1, commonalities 342
 LiS 2.18.1, compatibility 329
 LiS 2.18.1, compatibility functions 242, 342
 LiS 2.18.1, configuration 343
 LiS 2.18.1, differences 342
 LiS 2.18.1, extensions 233
 LiS 2.18.1, portability 342
 lis_adjmsg(9) 242
 lis_allocb(9) 242
 lis_allocb_physreq(9) 242
 lis_alloccq(9) 242
 lis_appq(9) 242, 342
 lis_backq(9) 242
 lis_bcanput(9) 243
 lis_bcanputnext(9) 243
 lis_bcopy(9) 243
 lis_bufcall(9) 243
 lis_bzero(9) 243
 lis_canenable(9) 243
 lis_canput(9) 243
 lis_canputnext(9) 243
 lis_cmn_err(9) 243
 lis_copyb(9) 243
 lis_copymsg(9) 243
 lis_datamsg(9) 243
 lis_date(9) 243
 lis_date(9) 343
 lis_dupb(9) 243
 lis_dupmsg(9) 243
 lis_enableok(9) 243
 lis_esballoc(9) 243
 lis_esbbcall(9) 243
 lis_esbbcall(9) 343

lis_find_strdev(9) 243
 lis_find_strdev(9) 343
 lis_flushband(9) 243
 lis_flushq(9) 243
 lis_freeb(9) 243
 lis_freemsg(9) 243
 lis_freeq(9) 243
 lis_getq(9) 243
 lis_insq(9) 243
 lis_isdatablk(9) 243
 lis_isdatamsg(9) 243
 lis_linkb(9) 243
 lis_mknod(9) 243
 lis_mknod(9) 343
 lis_mount(9) 243
 lis_mount(9) 343
 lis_msgdsize(9) 243
 lis_msgpullup(9) 243
 lis_msgsize(9) 243
 lis_noenable(9) 243
 lis_OTHER(9) 243
 lis_OTHER(9) 343
 lis_OTHERQ(9) 243
 lis_pullupmsg(9) 243
 lis_putbq(9) 243
 lis_putctl(9) 243
 lis_putctl(9) 243
 lis_putnext(9) 243
 lis_putnextctl(9) 243
 lis_putnextctl(9) 243
 lis_putq(9) 243
 lis_qattach(9) 244
 lis_qclose(9) 244
 lis_qdetach(9) 244
 lis_qenable(9) 244
 lis_qopen(9) 244
 lis_qprocsoff(9) 244
 lis_qprocson(9) 244
 lis_qreply(9) 244
 lis_qsize(9) 244
 lis_RD(9) 244
 lis_register_strdev(9) 242, 244
 lis_register_strdev(9) 343
 lis_register_strdrv(9) 25
 lis_register_strmod(9) 25, 242, 244
 lis_register_strmod(9) 343
 lis_rmvb(9) 244
 lis_rmvq(9) 244
 lis_safe_canenable(9) 244
 lis_safe_enableok(9) 244
 lis_safe_noenable(9) 244
 lis_safe_OTHERQ(9) 244
 lis_safe_putnext(9) 244
 lis_safe_qreply(9) 244
 lis_safe_RD(9) 244
 lis_safe_SAMESTR(9) 244

lis_safe_WR(9) 244
 lis_SAMESTR(9) 244
 lis_stream_utils(9) 244
 lis_strqget(9) 244
 lis_strqset(9) 244
 lis_testb(9) 244
 lis_timeout(9) 244
 lis_umount(9) 244
 lis_umount(9) 343
 lis_umount2(9) 244
 lis_umount2(9) 343
 lis_unbufcall(9) 244
 lis_unlink(9) 244
 lis_unlink(9) 343
 lis_unlinkb(9) 244
 lis_unregister_strdev(9) 242, 244
 lis_unregister_strdev(9) 343
 lis_unregister_strmod(9) 242, 244
 lis_unregsiter_strmod(9) 343
 lis_untimeout(9) 244
 lis_version(9) 244
 lis_version(9) 343
 lis_WR(9) 244
 lis_xmsgsize(9) 244
 lis_xmsgsize(9) 343
 LOCK(9) 234, 336
 LOCK_ALLOC(9) 234, 336
 LOCK_DEALLOC(9) 234, 336
 LOCK_OWNED(9) 234, 336
 log(4) 34, 35, 225
 log(4) 255, 321
 log(7) 305, 306, 321
 log, driver 225
 Log, driver 225
 log.h 223
 logging, STREAMS 34
 lstat(2) 250

M

M_BACKDONE 34
 M_BACKWASH 33
 M_BREAK .. 20, 33, 57, 126, 208, 209, 213, 214, 216, 281
 M_COPYIN .. 21, 34, 57, 113, 114, 121, 123, 132, 208, 278, 283, 289, 290, 292
 M_COPYOUT 21, 34, 57, 113, 114, 118, 123, 132, 208, 278, 283, 284, 289, 291, 292
 M_CTL 20, 23, 33, 57, 207, 208, 209, 210, 215, 216, 218, 281, 282, 293, 312
 M_DATA ... 20, 21, 24, 33, 54, 55, 57, 61, 62, 64, 75, 76, 88, 89, 90, 107, 110, 114, 116, 123, 131, 132, 143, 144, 147, 157, 166, 167, 179, 187, 190, 208, 209, 210, 214, 218, 281, 282, 283, 284, 286, 289, 290, 291, 292, 296, 298, 299, 301, 303

M_DELAY .. 20, 33, 57, 143, 167, 209, 216, 281, 282, 298, 299, 303
 M_DONTPLAY 34
 M_ERROR .. 21, 30, 34, 57, 65, 88, 99, 132, 166, 169, 190, 208, 236, 239, 290
 M_EVENT 33
 M_FLUSH 21, 24, 34, 57, 88, 124, 125, 126, 127, 128, 131, 133, 135, 143, 154, 157, 161, 166, 187, 190, 198, 208, 209, 218, 220, 290, 291
 M_HANGUP .. 21, 34, 57, 99, 103, 167, 168, 190, 195, 198, 208, 220, 291
 M_HPDATA 34
 M_IOCACK .. 21, 34, 57, 110, 111, 112, 113, 114, 132, 147, 168, 179, 208, 214, 216, 217, 283, 284, 289, 291, 292, 304
 M_IOCATA 21, 34, 57, 113, 114, 116, 121, 123, 132, 209, 279, 289, 292
 M_IOCNAK .. 21, 24, 34, 57, 110, 111, 113, 114, 116, 117, 132, 147, 168, 187, 208, 214, 216, 217, 283, 284, 289, 292, 304
 M_IOCTL 8, 20, 24, 33, 57, 109, 110, 111, 112, 113, 114, 115, 118, 120, 121, 132, 140, 147, 157, 161, 165, 168, 179, 180, 187, 209, 214, 216, 217, 218, 278, 282, 283, 284, 289, 291, 292, 304, 312
 M_LETSPLAY 34
 M_NOTIFY 34
 M_PASSFP 20, 33, 57, 284
 M_PCCTL 34
 M_PCEVENT 34
 M_PCPROTO ... 21, 24, 34, 57, 61, 62, 64, 75, 76, 86, 95, 99, 110, 143, 147, 167, 218, 285, 286, 293, 298, 299, 303
 M_PCRSE 21, 34, 57, 293
 M_PCSETOPTS 34
 M_PCSIG 21, 34, 57, 96, 99, 102, 103, 167, 208, 216, 293
 M_PROTO .. 20, 21, 24, 33, 57, 61, 62, 64, 75, 76, 86, 88, 110, 143, 147, 166, 167, 214, 218, 284, 285, 286, 293, 298, 299, 303, 312, 321
 M_READ 9, 21, 30, 34, 57, 209, 218, 287, 293
 M_RSE 21, 33, 57, 285
 M_SETOPTS .. 21, 33, 57, 58, 63, 102, 103, 132, 160, 169, 208, 209, 216, 217, 272, 273, 280, 285, 288, 289, 293
 M_SIG 21, 30, 33, 57, 61, 96, 99, 103, 167, 208, 288, 293
 M_START 21, 34, 58, 208, 209, 218, 293, 294
 M_STARTI 21, 34, 58, 208, 209, 218, 294
 M_STOP 21, 34, 58, 208, 209, 218, 293, 294
 M_STOPI 21, 34, 58, 208, 209, 218, 294
 M_TRAIL 33
 M_UNHANGUP 34
 MacOS 10
 major(9) 233, 337
 makedev(9) 233, 337

makedev(x, y) 137
makedevice(9)..... 231, 247, 248, 333
makedevice(x, y) 137
 Manipulating Modules 6
max(9) 231, 333
max(a, b) 136
MAX_CHRDEV 271
MAXINT 271, 272, 273
mbinfo(9) 228
mblock_t 296, 298
 mechanism 37
 mechanism overview 37
 message allocation and freeing 88
 Message Block 31
 message handling 229
 message queues and priority 64
 message structure 58
 messages 57
 messages overview 57
mi_bufcall(9) 227, 237, 251, 254, 337
mi_close_comm(9) 237, 337
mi_next_ptr(9) 237, 337
mi_open_comm(9) 238, 337
mi_prev_ptr(9) 238, 337
min(9) 231, 333
min(a, b) 136
minor(9) 233, 337
 miscellaneous functions 231
mknod(8) 36
mknod(9) 232, 247, 333
mod_info(9) 242, 341
mod_install(9) 242, 341
mod_remove(9) 242, 341
MODBLKSZ 89
modinfo(9) 227
MODOPEN 108, 217
 module and driver environment 105
 module entry points 227
 module example 245
 Module Information 28
 Module Statistics 28
 Module Switch Table 26
module_info 68
module_info(9) 28, 227
module_stat(9) 28, 228
 modules and drivers 105
 modules, connld 225
 modules, Pipe Module 225
 modules, pipemod 225
 modules, sc 225
 modules, sth 225
 modules, Stream Head 225
 modules, STREAMS 225
 modules, STREAMS Configuration 225
mount(8) 249
mount(9) 232, 333

MPSTR_QLOCK(9) 234, 336
MPSTR_QRELE(9) 234, 336
MPSTR_STPLOCK(9) 234, 336
MPSTR_STPRELE(9) 234, 336
MSG_ANY 63
MSG_BAND 62, 63
MSG_HIPRI 62, 63
msgb(9) 88
msgb(9) 228
msgdsize(9) 21, 229, 301, 331
MSGMARK 67
MSGNOLOOP 128
msgphysreq(9) 241, 340
msgpullup(9) 229, 331
msgpullup-physreq(9) 241, 340
msgscgth(9) 241, 340
msgsize(9) 233, 334
 multi-threading 251

N

named streams device 249
 Named STREAMS Device, driver 225
NDELAY 29
 networking, STREAMS 36
NLOGARGS 306
noenable(9) 23, 24, 74, 75, 230, 299, 301, 302, 303, 332
NOERROR 290
nsdev(4) 225, 249
 nsdev, driver 225
NSTRPUSH 44, 325
NULL 137
 Null STREAM, driver 225
nuls(4) 225
 nuls, driver 225

O

O_NDELAY 44, 98, 135, 197, 198, 203, 208, 220, 287, 288
O_NONBLOCK ... 44, 98, 135, 197, 198, 203, 220, 287, 288
 oddball functions 333
OK_ACK 82, 88
open(2) 16, 17, 31, 37, 44, 61, 102, 226
 OpenGroup 12
 OpenGroup Specifications 12
 OpenSS7 Project 3, 4
 OpenUnix 10
OSF/1 10, 25, 240, 247, 329, 338, 339
OSF/1 1.2/Digital UNIX 240, 338, 339
OSF/1 1.2/Digital UNIX, commonalities 339
OSF/1 1.2/Digital UNIX, compatibility 329

OSF/1 1.2/Digital UNIX, compatibility functions
 240, 339
 OSF/1 1.2/Digital UNIX, configuration 339
 OSF/1 1.2/Digital UNIX, differences 339
 OSF/1 1.2/Digital UNIX, portability 338
 OSF/1, compatibility module 240
 OSF/1, core functions 240
 OTHER(9) 157
 OTHERQ(9) 23, 129, 230, 302, 304, 307, 332
 OTHERQ(9) 343
 overview 15

P

p_pgrp 135
 p_pid 135
 param.h 106, 135, 137
 PCATCH 105, 135, 137
 pcmmsg(9) 230, 331
 Pipe Module, module 225
 pipe(2) 31, 35, 37, 43, 226
 pipe(4) 225
 pipe, driver 225
 Pipe, driver 225
 PIPE_BUF 197, 198
 pipemod(4) 225
 pipemod, module 225
 pipes, STREAMS 35
 poll(2) 31, 37, 95, 96, 98, 226
 poll.h 97
 POLL_ERR 100
 POLL_HUP 100
 POLL_IN 100
 POLL_MSG 100
 POLL_OUT 100
 POLL_PRI 100
 POLLERR 98, 290
 POLLHUP 98, 291
 POLLIN 95, 96, 97, 98
 polling and signalling 95
 POLLMSG 95, 96
 POLLNORM 95, 96
 POLLNVAL 98
 POLLOUT 95, 96, 98
 POLLPRI 95, 96
 POLLRDBAND 95, 96
 POLLRDNORM 95, 96
 POLLWRBAND 95, 96
 POLLWRNORM 95, 96
 portability 331
 porting, AIX 5L Version 5.1 337
 porting, core function support 331
 porting, HP-UX 11.0i v2 338
 porting, LiS 2.18.1 342
 porting, OSF/1 1.2/Digital UNIX 338
 porting, Solaris 9/SunOS 5.9 340

porting, Super/UX 342
 porting, SVR 4.2 MP 334
 porting, UnixWare 7.1.3 (OpenUnix 8) 339
 porting, UXP/V 342
 POSIX 12
 power(9) 242, 342
 printf(3) 306, 309
 printf(9) 241
 probe(9) 242, 342
 proc.h 132, 135
 processing routines 49
 pullupmsg(9) 21, 229, 302, 331
 put and service procedures 49
 put(9) 23, 24, 227, 230, 251, 253, 332, 338, 339
 putbq(9) 23, 69, 73, 74, 134, 141, 145, 230, 251,
 302, 332
 putctl(9) 23, 230, 236, 239, 303, 332
 putctl1(9) 23, 230, 303, 332
 putctl2(9) 232, 236, 239, 333, 337, 338
 puthere(9) 240, 339
 putmsg(2) 31, 37, 61, 62, 64, 75, 76, 80, 83, 84,
 225, 226, 281, 285, 286, 321
 putnext(9) 23, 24, 71, 73, 74, 109, 129, 133, 141,
 144, 145, 198, 203, 230, 251, 254, 303, 304,
 307, 332
 putnextctl(9) 230, 239, 332
 putnextctl1(9) 230, 332
 putnextctl2(9) 232, 236, 239, 333, 338
 PUTOUTDATA 123
 putpmsg(2) 31, 62, 225, 226, 273
 putq(9) 23, 24, 50, 55, 65, 69, 71, 72, 74, 133,
 145, 230, 251, 299, 302, 303, 304, 332
 PZERO 105, 137

Q

q_count 301
 q_lowat 301
 q_next 296
 qadmin 300
 qadmin(9) 227
 qattach(9) 232, 333
 QB_BACK 69
 qb_count 301
 QB_FULL 69
 qb_lowat 301
 QB_WANTW 69, 296
 QBACK 68
 qband 304
 qband(9) 228, 301, 304
 qbufcall(9) 227, 242, 251, 254, 341
 qclose(9) 227, 233, 251, 254, 333, 344
 QCOUNT 66
 qcountstrm(9) 233, 334
 qdetach(9) 233, 333
 QENAB 68

Index

qenable(9) .. 23, 24, 75, 92, 189, 230, 300, 304, 332
QFIRST .. 66
QFLAG .. 66
QFULL .. 68, 145, 209
QHLIST .. 68, 236
qi_putp() .. 304
qinit(9) .. 28, 228, 304
QLAST .. 66
QNOENAB .. 303
QNOENB .. 68
QOLD .. 68
qopen(9) ... 227, 232, 247, 248, 250, 251, 253, 333
qopen(9) .. 344
qprocsoff(9) .. 230, 254, 332, 344
qprocson(9) .. 230, 253, 254, 332, 344
QREADR .. 68
qreply(9) .. 23, 24, 112, 230, 251, 254, 283, 304, 332
qsize(9) .. 23, 230, 304, 332
qtimeout(9) .. 227, 242, 251, 254, 341
queclass(9) .. 242, 341
queinfo(9) .. 228
Queue .. 30
Queue Band .. 30
queue handling .. 230
Queue Initialization .. 27
queue(9) .. 228
qunbufcall(9) .. 242, 254, 341
quntimeout(9) .. 242, 254, 341
QUSE .. 68
qwait(9) .. 242, 341
qwait_sig(9) .. 242, 341
QWANTR .. 68, 74, 300, 303
QWANTW .. 68, 141, 145, 296
qwriter(9) .. 242, 254, 341

R

RD(9) .. 23, 129, 157, 230, 304, 307, 332
read(2) ... 21, 31, 37, 61, 63, 64, 67, 75, 226, 286,
287
readv(2) .. 31, 226
Realities of STREAMS .. 10
reference .. 223
register_clone(9) .. 228
register_cmajor(9) .. 228
register_strdev(9) .. 228
register_strdrv(9) .. 25, 26, 228
register_strmod(9) .. 25, 27, 229
register_strnod(9) .. 229
registration .. 228
request_module(9) .. 249, 250
rmalloc(9) .. 236, 336
rmalloc(9) .. 343
rmalloc(mp, size) .. 136
rmalloc_wait(9) .. 236, 337
rmalloc_wait(9) .. 343

rmallocmap(9) .. 236, 336
rmallocmap(9) .. 343
rmallocmap_wait(9) .. 236, 337
rmfree(9) .. 236, 337
rmfree(9) .. 343
rmfree(mp, size, i) .. 136
rmfreemap(9) .. 236, 337
rmfreemap(9) .. 343
rmget(9) .. 236, 337
rminit(9) .. 236, 337
rminit(9) .. 343
rminit(mp, mapsize) .. 136
rmsetwant(9) .. 236, 337
rmsetwant(9) .. 343
RMSGD .. 64, 286
RMSGN .. 64, 207, 286
rmvb(9) .. 229, 305, 331
rmvq(9) .. 23, 145, 230, 305, 332
rmwanted(9) .. 236, 337
rmwanted(9) .. 343
RNORM .. 63, 207, 286
RPROTDAT .. 64, 286
RPROTDIS .. 64, 286
RPROTNORM .. 64, 286
RS_HIPRI .. 62, 81, 82
RSLEEP .. 30
runqueues(9) .. 271
RW_ALLOC(9) .. 234, 336
RW_DEALLOC(9) .. 234, 336
RW_RDLOCK(9) .. 234, 336
RW_TRYRDLOCK(9) .. 234, 336
RW_TRYWRLOCK(9) .. 234, 336
RW_UNLOCK(9) .. 235, 336
RW_WRLOCK(9) .. 235, 336

S

S_BANDURG .. 99
S_ERROR .. 99, 100
S_HANGUP .. 99, 100
S_HIPRI .. 99, 100
S_INPUT .. 99, 100
S_MSG .. 99, 100
S_OUTPUT .. 99, 100
S_RDBAND .. 99
S_RDNORM .. 99
S_WRBAND .. 99
S_WRNORM .. 99
sad(4) .. 35, 225
sad, driver .. 225
sad.h .. 223
SAMESTR(9) .. 230, 332
SAP_ALL .. 327
SAP_CLEAR .. 327
SAP_ONE .. 327
SAP_RANGE .. 327

- sc(4) 225
- sc, module 225
- scls(8) 35
- scls(8) 247, 255
- scls(8) 260
- seinfo(9) 228
- select(2) 31, 226
- service interfaces 75
- SET_ADDR 116
- SET_OPTIONS 112
- setpgid(2) 100
- setpgrp(2) 100
- setq(9) 232, 333
- setsid(2) 100
- shinfo(9) 228
- sigaction(2) 100
- SIGCONT 101
- SIGHUP 103, 167, 291
- siginfo.h 100
- SIGINT 220
- signal(2) 95, 99, 100, 226
- signal(5) 100, 101
- signal.h 135, 220
- SIGPOLL 29, 95, 96, 98, 99, 100, 288, 289
- sigsend(2) 100
- SIGSTOP 101
- SIGTSTP 101, 102
- SIGTTIN 101, 103
- SIGTTOU 102, 103
- SIGURG 99
- SIGWINCH 217
- SIOCSPGRP 289
- SL_CONSOLE 306
- SL_ERROR 305
- SL_FATAL 305
- SL_NOTE 306
- SL_NOTIFY 306
- SL_TRACE 305
- SL_WARN 306
- sleep(9) 88, 139, 233, 334
- sleep(chan, pri) 136
- SLEEP_ALLOC(9) 235, 336
- SLEEP_DEALLOC(9) 235, 336
- SLEEP_LOCK(9) 235, 336
- SLEEP_LOCK_SIG(9) 235, 336
- SLEEP_LOCKAVAIL(9) 235, 336
- SLEEP_LOCKOWNED(9) 235, 336
- SLEEP_TRYLOCK(9) 235, 336
- SLEEP_UNLOCK(9) 235, 336
- SNDHOLD 273
- SNDMREAD 30
- SNDZERO 197
- SO_ALL 29, 286
- SO_BAND 30, 288
- SO_COPYOPT 30
- SO_DELIM 30
- SO_ERROPT 30
- SO_HIWAT 29, 287, 288
- SO_ISNTTY 30, 102, 288
- SO_ISTTY 29, 102, 132, 169, 208, 288, 289
- SO_LOOP 30
- SO_LOWAT 29, 287, 288
- SO_MAXBLK 30
- SO_MAXPSZ 29, 286
- SO_MINPSZ 29, 286
- SO_MREADOFF 29, 287, 293
- SO_MREADON 29, 287, 293
- SO_NDELOFF 29, 287, 288
- SO_NDELON 29, 208, 287, 288
- SO_NODELIM 30
- SO_READOPT 29, 286
- SO_STRHOLD 30
- SO_TONSTOP 30, 103, 288
- SO_TOSTOP 30, 103, 288
- SO_WROFF 29, 286
- sockets 4
- Solaris 10, 25, 241, 247, 329, 340, 342
- Solaris 9/SunOS 5.9 241, 340, 342
- Solaris 9/SunOS 5.9, commonalities 340
- Solaris 9/SunOS 5.9, compatibility 329
- Solaris 9/SunOS 5.9, compatibility functions 241, 340
- Solaris 9/SunOS 5.9, configuration 342
- Solaris 9/SunOS 5.9, differences 340
- Solaris 9/SunOS 5.9, portability 340
- Solaris, compatibility module 241
- Solaris, core functions 241
- spec file system 249
- specfs(5) 249, 250
- spefs(5) 249
- spl(9) 235, 335
- spl0(9) 235, 335
- spl1(9) 235, 335
- spl2(9) 235, 335
- spl3(9) 235, 335
- spl4(9) 235, 335
- spl5(9) 235, 335
- spl7(9) 235, 335
- splhi(9) 309
- spln() 136
- splstr() 136
- splstr(9) 129, 132, 237, 305, 307, 337
- splx(9) 235, 237, 305, 335, 337
- sponsors 1
- spx(4) 35
- SQLVL_DEFAULT 251
- SQLVL_ELSEWHERE 252
- SQLVL_GLOBAL 252
- SQLVL_MODULE 251, 252
- SQLVL_NOP 253
- SQLVL_QUEUE 253
- SQLVL_QUEUEPAIR 252

- srv(9) 227, 251, 253
- Standardized Service Interfaces 6
- stat(2) 250
- stdat(9) 228
- STFROZEN 30
- sth(4) 225
- sth, module 225
- STPLEX 30
- str_install(9) 238, 239, 267, 338
- str_install_AIX(9) 25
- str_install_HPUX(9) 25
- str_uninstall(9) 25, 240, 338
- strace(1) 321
- strace(8) 34, 35
- strace(8) 255
- strace(8) 261
- strapush(9) 228
- strchg(1) 35
- strchg(1) 123, 124
- strclean(1) 321
- strclean(8) 255
- strclean(8) 262
- STRCLOSE 30
- strconf(1) 35
- strconf(1) 123, 124
- strconf(8) 255
- strconf(8) 263
- strconf_t 25
- STRCTLSZ 326
- STRDELIM 30
- STRDERR 30
- Stream Administration 25
- stream as controlling terminal 100
- stream construction 38
- Stream Head 29
- Stream Head, module 225
- Stream Table 27
- stream.h 106, 135
- stream_inst(9) 25
- streamadm(9) 25, 228
- streamio(7) 17
- streamio(7) 37, 44, 47, 48, 63, 98, 109, 110, 111, 112, 118, 124, 271, 272, 273, 279, 282, 284, 344
- STREAMS Administrative Driver, driver 225
- STREAMS configuration 323
- STREAMS Configuration, module 225
- STREAMS data structures 275
- STREAMS debugging 309
- STREAMS drivers 149
- STREAMS message types 281
- STREAMS modules 139
- STREAMS multiplexing 171
- STREAMS utilities 295
- STREAMS versus Sockets 4
- STREAMS, administration 255
- STREAMS, administrative driver 35
- STREAMS, application interface 31
- STREAMS, benefits 6
- STREAMS, configuration 251
- STREAMS, definitions 15
- STREAMS, drivers 225
- STREAMS, FIFOs 36
- STREAMS, input-output controls 226
- STREAMS, kernel level facilities 32
- STREAMS, logging 34
- STREAMS, modules 225
- STREAMS, networking 36
- STREAMS, pipes 35
- STREAMS, subsystems 34
- STREAMS, terminal i/o 35
- STREAMS, what is it? 3
- streams-0.7a.3 3
- STREAMS-based pipes and fifos 195
- STREAMS-based terminal subsystem 205
- streams_close_comm(9) 240, 339
- streams_get_sleep_lock(9) 240, 338
- streams_mknod(8) 255
- streams_mknod(8) 264
- streams_open_comm(9) 240, 339
- streams_open_ocomm(9) 240, 339
- streams_put(9) 240, 338
- streamtab 40, 43, 68, 106, 107, 130, 152, 156, 178, 179, 184
- streamtab 275
- streamtab 276
- streamtab(9) 26, 27, 40, 43, 105, 106, 228, 248, 250
- strerr(1) 321
- strerr(8) 34, 35
- strerr(8) 255
- strerr(8) 265
- strevent(9) 228
- STRHOLD 30
- STRHUP 30
- strinfo(8) 35
- strinfo(8) 247, 255
- strinfo(8) 266
- striocall(9) 241, 340
- STRISFIFO 30
- STRISPIPE 30
- STRISOCK 30
- STRISTTY 30
- strload(8) 35
- strload(8) 247, 255
- strload(8) 267
- strlog(9) 34, 35, 232, 305, 312, 321, 333
- strlog.h 223
- strmod_add(9) 25, 240, 339
- strmod_del(9) 25, 240, 339
- STRMSGSZ 64, 325
- STRMSIG 30
- stropts.h 135, 223

- STRPRI 30
- strqget(9) 23, 66, 69, 231, 306, 332
- strqset(9) 66, 69, 231, 306, 332
- strreset(1) 35
- strsetup(8) 35
- strsetup(8) 247, 255
- strsetup(8) 269
- STRTOSTOP 30
- structures 227
- strvf(8) 255
- strvf(8) 270
- STWOPEN 30
- STWRERR 30
- subsystems, STREAMS 34
- SunOS 10, 241, 340, 342
- Super-UX 10
- Super/UX 247, 329, 342
- Super/UX, commonalities 342
- Super/UX, compatibility 329
- Super/UX, compatibility functions 342
- Super/UX, configuration 342
- Super/UX, differences 342
- Super/UX, portability 342
- suser(9) 108
- SUSP 102
- SV_ALLOC(9) 235, 336
- SV_BROADCAST(9) 235, 336
- SV_DEALLOC(9) 235, 336
- SV_SIGNAL(9) 235, 336
- SV_WAIT(9) 235, 336
- SV_WAIT_SIG(9) 235, 336
- SVR 4 11
- SVR 4.2 247, 339
- SVR 4.2 MP 3, 11, 12, 232, 233, 234, 235, 236, 237, 238, 239, 240, 251, 329, 331, 334, 337, 339, 341
- SVR 4.2 MP DDI/DKI 247, 251
- SVR 4.2 MP, commonalities 334
- SVR 4.2 MP, compatibility 329
- SVR 4.2 MP, compatibility functions 233, 334
- SVR 4.2 MP, compatibility module 233
- SVR 4.2 MP, configuration 337
- SVR 4.2 MP, core functions 233
- SVR 4.2 MP, differences 334
- SVR 4.2 MP, portability 334
- SVR 4.2 MP, *STREAMS* 12
- SVR4 343
- synchronization, default 251
- synchronization, elsewhere 252
- synchronization, global 252
- synchronization, module 252
- synchronization, none 253
- synchronization, queue 253
- synchronization, queue pair 252
- synchronous callbacks 254
- synchronous callouts 254
- synchronous entry points 253
- sys.streams.cltime 271
- sys.streams.hiwat 272
- sys.streams.lowat 272
- sys.streams.max_apush 271
- sys.streams.max_mblk 271
- sys.streams.max_stramod 271
- sys.streams.max_strdev 271
- sys.streams.max_stmod 271
- sys.streams.maxpsz 272
- sys.streams.minpsz 272
- sys.streams.msg_priority 271
- sys.streams.nband 272
- sys.streams.nstrmsgs 272
- sys.streams.nstrpush 272
- sys.streams.reuse_fmodsw 273
- sys.streams.rtime 273
- sys.streams.strctlsz 273
- sys.streams.strhold 273
- sys.streams.strmsgsz 273
- sys.streams.strthresh 273
- sys/aixddi.h 224
- sys/cmn_err.h 224
- sys/conf.h 124
- sys/ddi.h 224, 307
- sys/debug.h 224
- sys/dki.h 224
- sys/file.h 108
- sys/hpuxddi.h 224
- sys/kmem.h 224
- sys/lisddi.h 224
- sys/log.h 223, 305
- sys/osfddi.h 224
- sys/sad.h 223
- sys/sc.h 223
- sys/spec_fs_i.h 224
- sys/strconf.h 25, 223
- sys/strdebug.h 223
- sys/stream.h 57, 110, 114, 223, 275, 282, 283, 299, 307
- sys/streams/config.h 224
- sys/strlog.h 223, 305
- sys/stropts.h 47, 142, 143, 223
- sys/strsubr.h 223
- sys/sunddi.h 224
- sys/svr4ddi.h 224
- sys/uv7ddi.h 224
- SYS_CFGDD 267
- SYS_CFGMOD 267
- sysconfig(9) 267
- sysctl(2) 271
- sysctl(8) 271
- sysmacros.h 135, 136
- system calls 225
- system controls 271
- system.h 137

T

TAB3 210
 TCFLSH 209
 TCGETA 111, 209, 214, 215, 216
 TCGETS 209, 214, 215, 216
 TCP/IP 4
 TCSBRK 209, 213, 214, 216
 TCSETA 102, 111, 209, 214, 216
 TCSETAF 102, 209, 214, 216
 TCSETAW 102, 209, 214, 216
 TCSETS 209, 214, 216
 TCSETSF 209, 214, 216
 TCSETSW 209, 214, 216
 TCXONC 209
 terminal i/o, STREAMS 35
 termio(7) 100, 102, 111
 termios(2) 100
 termiox.h 213
 testb(9) 229, 307, 331
 time 137
 time(9) 240, 339
 timeout(9) 92, 227, 231, 251, 253, 333, 344
 timeout(func, arg, ticks) 136
 TIOCGWINSZ 209, 214, 216, 217
 TIOCREMOTE 216
 TIOCSIGNAL 216
 TIOCSGRP 289
 TIOCSWINSZ 209, 214, 216
 TOSTOP 102, 103
 TRANSPARENT 113, 114, 116
 TRUE 298
 TRYLOCK(9) 234, 336
 tty.h 208
 types.h 106, 135, 137

U

u_procp 135
 u_ttyp 135
 umount(9) 232, 333
 unbufcall(9) 21, 91, 229, 307, 331
 unfreezestr(9) 231, 254, 332
 UNITDATA_IND 84, 86, 88
 UNITDATA_REQ 88
 UNIX 03 compliance 12
 UNIX 95 compliance 13
 UNIX 98 compliance 13
 UNIX System V Release 3.0 3
 UNIX System V Release 4 3
 UNIX System V Release 4.2 3, 254
 UnixWare 10, 25, 240, 241, 247, 329, 339, 340
 UnixWare 7.1.3 (OpenUnix 8) 240, 339, 340
 UnixWare 7.1.3 (OpenUnix 8), commonalities 339
 UnixWare 7.1.3 (OpenUnix 8), compatibility .. 329

UnixWare 7.1.3 (OpenUnix 8), compatibility
 functions 240, 339
 UnixWare 7.1.3 (OpenUnix 8), configuration .. 340
 UnixWare 7.1.3 (OpenUnix 8), differences 339
 UnixWare 7.1.3 (OpenUnix 8), portability 339
 UnixWare, compatibility module 241
 UnixWare, core functions 240
 unlink(9) 232, 247, 333
 unlinkb(9) 21, 229, 307, 331
 UNLKPT 220
 UNLOCK(9) 234, 336
 unregister_clone(9) 228
 unregister_cmajor(9) 228
 unregister_strdev(9) 228
 unregister_strdrv(9) 229
 unregister_strmod(9) 229
 unregister_strnod(9) 229
 untimout(9) 231, 333
 untimout(id) 136
 unweldq(9) .. 232, 237, 239, 240, 333, 337, 338, 339
 User Credentials 31
 user.h 132, 135
 UXP/V 10, 247, 329, 342
 UXP/V, commonalities 342
 UXP/V, compatibility 329
 UXP/V, compatibility functions 342
 UXP/V, configuration 342
 UXP/V, differences 342
 UXP/V, portability 342

V

VMIN 209
 VTIME 209
 vtop(9) 233, 334
 vtop(vaddr, NULL) 136

W

waitid(2) 100
 waitpid(3C) 100
 wakeup(9) 233, 334
 wakeup(chan) 136
 wantio(9) 238, 338
 wantmsg(9) 238, 338
 weldq(9) 232, 237, 239, 240, 333, 337, 338, 339
 what is *STREAMS*? 3
 Why Compatibility? 11
 Why Fast? 10
 Why Linux? 11
 Why *STREAMS*? 4
 WR(9) 23, 129, 157, 230, 307, 332
 write(2) .. 31, 37, 55, 61, 64, 75, 226, 273, 286, 287
 writev(2) 31, 226, 273
 WSLEEP 30

X	
XCASE.....	47
XENIX.....	340
xmsgsize(9).....	233, 334

