OpenSS7 STREAMS Programmer's Guide

OpenSS7
STREAMS Programmer's Guide

Version 1.1 Edition 7.20141001
Updated October 25, 2014
Distributed with Package openss7-1.1.7.20141001

Copyright © 2008-2014 Monavacon Limited
All Rights Reserved.

Abstract:

This document is a STREAMS Programmer's Guide containing technical details concerning the implementation of the OpenSS7 for OpenSS7. It contains recommendations on software architecture as well as platform and system applicability of the OpenSS7.

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

Published by:

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

Copyright © 2008-2014 Monavacon Limited
Copyright © 2001-2008 OpenSS7 Corporation
Copyright © 1997-2000 Brian F. G. Bidulock

All Rights Reserved.

Unauthorized distribution or duplication is prohibited.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled Section H.2 [GNU Free Documentation License], page 156.

Permission to use, copy and distribute this documentation without modification, for any purpose and without fee or royalty is hereby granted, provided that both the above copyright notice and this permission notice appears in all copies and that the name of *OpenSS7 Corporation* not be used in advertising or publicity pertaining to distribution of this documentation or its contents without specific, written prior permission. *OpenSS7 Corporation* makes no representation about the suitability of this documentation for any purpose. It is provided "as is" without express or implied warranty.

Notice:

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

Short Contents

Preface 3
1 Introduction 13
2 Overview 39
3 Mechanism 41
4 Processing 57
5 Messages 61
6 Polling 87
7 Modules and Drivers 89
8 Modules 91
9 Drivers 93
10 Multiplexing 95
11 Pipes and FIFOs 97
12 Terminal Subsystem 99
13 Synchronization 101
14 Reference 111
15 Conformance 113
16 Portability 115
17 Developing Portable STREAMS Modules 127
A Data Structures 131
B Message Types 133
C Utilities 135
D Debugging 137
E Configuration 139
F Administration 141
G Examples 143
H Copying 145
Glossary 163
Index 169

Table of Contents

Preface . . . 3
Acknowledgements . . . 3
Sponsors . . . 3
Contributors . . . 3
Supporters . . . 3
Telecommunications . . . 3
Aerospace and Military . . . 5
Financial, Business and Security . . . 5
Education, Health Care and Nuclear Power . . . 6
Agencies . . . 6
Authors . . . 6
Maintainer . . . 6
Document Information . . . 7
Notice . . . 7
Abstract . . . 7
Objective . . . 7
Intent . . . 7
Audience . . . 7
Revisions . . . 7
Version Control . . . 8
ISO 9000 Compliance . . . 8
Disclaimer . . . 8
U.S. Government Restricted Rights . . . 8
Organization . . . 8
Conventions Used . . . 10
Other Documentation . . . 11
UNIX Edition . . . 11
Related Manuals . . . 11
Copyright . . . 11

1 Introduction . . . 13
1.1 Background . . . 13
1.2 What is STREAMS? . . . 14
1.2.1 Characteristics . . . 14
1.2.2 Components . . . 15
1.2.2.1 Stream head . . . 16
1.2.2.2 Module . . . 17
1.2.2.3 Driver . . . 18
1.2.2.4 Queues . . . 18
1.2.2.5 Messages . . . 18
1.3 Basic Streams Operations . . . 19
1.3.1 Basic Operations Example . . . 20
1.4 Components . . . 21
1.4.1 Queues . . . 21
1.4.1.1 Queue Procedures . . . 22
1.4.2 Messages . . . 22
1.4.2.1 Message Types . . . 23

1.4.2.2 Message Linkage 23
1.4.2.3 Message Queueing Priority 25
1.4.3 Modules 26
1.4.4 Drivers 28
1.4.5 Stream Head 29
1.5 Multiplexing 29
1.5.1 Fan-Out Multiplexers 31
1.5.2 Fan-In Multiplexers 32
1.5.3 Complex Multiplexers 33
1.6 Benefits of STREAMS 33
1.6.1 Standardized Service Interfaces 33
1.6.2 Manipulating Modules 34
1.6.2.1 Protocol Portability 34
1.6.2.2 Protocol Substitution 35
1.6.2.3 Protocol Migration 36
1.6.2.4 Module Reusability 37

2 Overview 39
2.1 Definitions 39
2.2 Concepts 39
2.3 Application Interface 39
2.4 Kernel Level Facilities 39
2.5 Subsystems 39

3 Mechanism 41
3.1 Mechanism Overview 41
3.1.1 STREAMS System Calls 42
3.2 Stream Construction 42
3.2.1 Opening a STREAMS Device File 45
3.2.1.1 First Open of a Stream 46
3.2.1.2 Subsequent Open of a Stream 47
3.2.2 Opening a STREAMS-based FIFO 47
3.2.3 Creating a STREAMS-based Pipe 49
3.2.4 Adding and Removing Modules 50
3.2.4.1 Pushing Modules 50
3.2.4.2 Popping Modules 50
3.2.5 Closing the Stream 51
3.2.6 Stream Construction Example 51
3.2.6.1 Inserting Modules 52
3.2.6.2 Module and Driver Control 54
3.2.6.3 Stream Dismantling with Modules 55
3.2.6.4 Stream Construction Example Summary 56

4 Processing 57
4.1 Procedures 57
4.1.1 Put Procedure 57
4.1.2 Service Procedure 59
4.1.3 Put and Service Procedure Summary 60
4.2 Asynchronous Example 60

5 Messages 61
5.1 Messages Overview 61
5.1.1 Message Types 61
5.1.1.1 Ordinary Messages 62
5.1.1.2 High Priority Messages 62
5.1.2 Expedited Data 63
5.2 Message Structure 63
5.2.1 Message Linkage 66
5.2.2 Sending and Receiving Messages 68
5.2.2.1 putmsg(2s) 69
5.2.2.2 getmsg(2s) 70
5.2.2.3 putpmsg(2s) 70
5.2.2.4 getpmsg(2s) 71
5.2.3 Control of Stream Head Processing 72
5.2.3.1 Read Options 72
5.2.3.2 Read Mode 72
5.2.3.3 Read Protocol 73
5.2.3.4 Write Options 73
5.2.3.5 Write Offset 74
5.3 Queues and Priority 74
5.3.1 Queue Priority Utilities 75
5.3.1.1 strqget(9) 76
5.3.1.2 strqset(9) 77
5.3.2 Queue Priority Commands 78
5.3.2.1 `I_FLUSHBAND` 78
5.3.2.2 `I_CKBAND` 79
5.3.2.3 `I_GETBAND` 79
5.3.2.4 `I_CANPUT` 79
5.3.2.5 `I_ATMARK` 80
5.3.2.6 `I_GETSIG` 81
5.3.2.7 `I_SETSIG` 81
5.3.3 The `queue` Structure 82
5.3.3.1 Using `queue` Information 83
5.3.3.2 `queue` Flags 83
5.3.4 The `qband` Structure 84
5.3.4.1 Using `qband` Information 84
5.3.5 Message Processing 84
5.3.5.1 Flow Control 84
5.3.6 Scheduling 84
5.3.6.1 Flow Control Variables 84
5.3.6.2 Flow Control Procedures 84
5.3.6.3 The STREAMS Scheduler 84
5.4 Service Interfaces 84
5.4.1 Service Interface Benefits 85
5.4.2 Service Interface Library Example 85
5.4.2.1 Accessing the Service Provider 85
5.4.2.2 Closing the Service Provider 85
5.4.2.3 Sending Data to the Service Provider 85
5.4.2.4 Receiving Data 85
5.4.2.5 Module Service Interface Example 85
5.5 Message Allocation 85
5.5.1 Recovering From No Buffers 85
5.6 Extended Buffers 85

6 Polling 87
6.1 Input and Output Polling 87
6.2 Controlling Terminal 87

7 Modules and Drivers 89
7.1 Environment 89
7.2 Input-Output Control 89
7.3 Flush Handling 89
7.4 Driver-Kernel Interface 89
7.5 Design Guidelines 89

8 Modules 91
8.1 Module 91
8.2 Module Flow Control 91
8.3 Module Design Guidelines 91

9 Drivers 93
9.1 External Device Numbers 93
9.2 Internal Device Numbers 93
9.3 spec File System 93
9.4 Clone Device 93
9.5 Named STREAMS Device 93
9.6 Driver 93
9.7 Cloning 93
9.8 Loop-Around Driver 93
9.9 Driver Design Guidelines 93

10 Multiplexing 95
10.1 Multiplexors 95
10.2 Connecting and Disconnecting Lower Stream 95
10.3 Multiplexor Construction Example 95
10.4 Multiplexing Driver 95
10.5 Persistent Links 95
10.6 Multiplexing Driver Design Guidelines 95

11 Pipes and FIFOs 97
11.1 Pipes and FIFOs 97
11.2 Flushing Pipes and FIFOs 97
11.3 Named Streams 97
11.4 Unique Connections 97

12 Terminal Subsystem 99
12.1 Terminal Subsystem 99
12.2 Pseudo-Terminal Subsystem 99

13 Synchronization 101
13.1 MT Configuration 102
13.2 Synchronous Entry Points 103
13.3 Synchronous Callbacks 104
13.4 Synchronous Callouts 104
13.5 Asynchronous Entry Points 104
13.6 Asynchronous Callbacks 104
13.7 Asynchronous Callouts 104
13.8 STREAMS Framework Integrity 104
13.9 MP Message Ordering 105
13.10 MP-UNSAFE Modules 105
13.10.1 MP-UNSAFE Open and Close Routines 105
13.10.2 MP-UNSAFE Put and Service Procedures 105
13.10.3 MP-UNSAFE Interrupt Service Routines 105
13.10.4 MP-UNSAFE Shared Data Structures 106
13.10.5 MP-UNSAFE Sleeping 106
13.11 MP-SAFE Modules 106
13.11.1 MP Put and Service Procedures 106
13.11.2 MP Timeout and Buffer Callbacks 106
13.11.3 MP Open and Close Procedures 107
13.11.4 MP Module Unloading 107
13.11.5 MP Locking 108
13.11.6 MP Asynchronous Callbacks 108
13.12 Stream Integrity 109

14 Reference 111
14.1 Files 111
14.2 System Modules 111
14.3 System Drivers 111
14.4 System Calls 111
14.5 Input-Output Controls 111
14.6 Module Entry Points 111
14.7 Structures 111
14.8 Registration 111
14.9 Message Handling 111
14.10 Queue Handling 111
14.11 Miscellaneous Functions 111
14.12 Extensions 111
14.13 Compatibility 111

15 Conformance 113
15.1 SVR 4.2 Compatibility 113
15.2 AIX Compatibility 113
15.3 HP-UX Compatibility 113
15.4 OSF/1 Compatibility 113
15.5 UnixWare Compatibility 113
15.6 Solaris Compatibility 113
15.7 SUX Compatibility 113
15.8 UXP Compatibility 113

16 Portability . . . 115

16.1 Core Function Support . . . 115
16.1.1 Core Message Functions . . . 115
16.1.2 Core UP Queue Functions . . . 115
16.1.3 Core MP Queue Functions . . . 116
16.1.4 Core DDI/DKI Functions . . . 116
16.1.5 Some Common Extension Functions . . . 117
16.1.6 Some Internal Functions . . . 117
16.1.7 Some Oddball Functions . . . 117
16.2 SVR 4.2 Portability . . . 117
16.2.1 Differences from SVR 4.2 MP . . . 117
16.2.2 Commonalities with SVR 4.2 MP . . . 118
16.2.3 Compatibility functions for SVR 4.2 MP . . . 118
16.2.3.1 Priority Levels . . . 118
16.2.3.2 Atomic Integers . . . 119
16.2.3.3 Basic Locks . . . 119
16.2.3.4 STREAMS Locks . . . 119
16.2.3.5 Read/Write Locks . . . 119
16.2.3.6 Sleep Locks . . . 119
16.2.3.7 Synchronization Variables . . . 119
16.2.3.8 Resource Allocation . . . 120
16.2.3.9 Device Numbering . . . 120
16.2.4 Configuration ala SVR 4.2 MP . . . 120
16.3 AIX Portability . . . 120
16.3.1 Differences from AIX 5L Version 5.1 . . . 120
16.3.2 Commonalities with AIX 5L Version 5.1 . . . 120
16.3.3 Compatibility Functions for AIX 5L Version 5.1 . . . 120
16.3.3.1 Core Extensions . . . 120
16.3.3.2 Common Module Utilities . . . 120
16.3.3.3 Registration . . . 121
16.3.3.4 Message Filtering . . . 121
16.3.4 Configuration ala AIX 5L Version 5.1 . . . 121
16.4 HP-UX Portability . . . 121
16.4.1 Differences from HP-UX 11.0i v2 . . . 121
16.4.2 Commonalities with HP-UX 11.0i v2 . . . 121
16.4.3 Compatibility Functions for HP-UX 11.0i v2 . . . 121
16.4.3.1 Core Extensions . . . 121
16.4.3.2 Registration . . . 121
16.4.3.3 Sleeping . . . 121
16.4.4 Configuration ala HP-UX 11.0i v2 . . . 121
16.5 OSF/1 Portability . . . 122
16.5.1 Differences from OSF/1 1.2/Digital UNIX . . . 122
16.5.2 Commonalities with OSF/1 1.2/Digital UNIX . . . 122
16.5.3 Compatibility Functions for OSF/1 1.2/Digital UNIX . . . 122
16.5.3.1 Core Extensions . . . 122
16.5.3.2 Common Module Utilities . . . 122
16.5.3.3 Registration . . . 122
16.5.3.4 Others . . . 122
16.5.4 Configuration ala OSF/1 1.2/Digital UNIX . . . 122
16.6 UnixWare Portability . . . 122
16.6.1 Differences from UnixWare 7.1.3 (OpenUnix 8) . . . 122
16.6.2 Commonalities with UnixWare 7.1.3 (OpenUnix 8) . . . 122
16.6.3 Compatibility Functions for UnixWare 7.1.3 (OpenUnix 8) . . . 122

16.6.3.1 Device Numbering . . . 123
16.6.3.2 Memory Alignment . . . 123
16.6.3.3 Direct STREAMS Input-Output Controls . . . 123
16.6.4 Configuration ala UnixWare 7.1.3 (OpenUnix 8) . . . 123
16.7 Solaris Portability . . . 123
16.7.1 Differences from Solaris 9/SunOS 5.9 . . . 123
16.7.2 Commonalities with Solaris 9/SunOS 5.9 . . . 123
16.7.3 Compatibility Functions for Solaris 9/SunOS 5.9 . . . 123
16.7.3.1 STREAMS Queue Referenced Callbacks . . . 123
16.7.3.2 STREAMS Registration . . . 124
16.7.3.3 DDI . . . 124
16.7.3.4 Loadable Module Interface . . . 124
16.7.4 Configuration ala Solaris 9/SunOS 5.9 . . . 125
16.8 SUX Portability . . . 125
16.8.1 Differences from Super/UX . . . 125
16.8.2 Commonalities with Super/UX . . . 125
16.8.3 Compatibility Functions for Super/UX . . . 125
16.8.4 Configuration ala Super/UX . . . 125
16.9 UXP Portability . . . 125
16.9.1 Differences from UXP/V . . . 125
16.9.2 Commonalities with UXP/V . . . 125
16.9.3 Compatibility Functions for UXP/V . . . 125
16.9.4 Configuration ala UXP/V . . . 125
16.9.4.1 Extensions . . . 125
16.9.4.2 Device Creation and Deletion . . . 125
16.9.4.3 Registration . . . 126

17 Developing Portable STREAMS Modules . . . 127

17.1 Memory Allocation . . . 127
17.2 Alignment of Message Buffers . . . 127
17.3 Disabling and Enabling Queue Procedures . . . 127
17.4 Freezing and Unfreezing Streams . . . 127
17.5 Passing Messages from Interrupt Service Routines . . . 127
17.6 Timeout Call Back and Link Identifiers . . . 127
17.7 Synchronization with Timeouts and Callback Functions . . . 127
17.8 Synchronization with Callout Functions . . . 128
17.9 Synchronization of Drivers and Modules . . . 128
17.10 Special STREAMS Message Types . . . 128
17.11 Use of Message Allocation Priorities . . . 128
17.12 Registration and Deregistration . . . 128
17.13 Device Numbering . . . 128
17.13.1 UNIX Device Numbering . . . 128
17.13.2 Linux Device Numbering . . . 128
17.13.3 OpenSS7 Device Numbering . . . 128

Appendix A Data Structures . . . 131

A.1 Stream Structures . . . 131
A.2 Queue Structures . . . 131
A.3 Message Structures . . . 131
A.4 Input Output Control Structures . . . 131
A.5 Link Structures . . . 131
A.6 Options Structures . . . 131

Appendix B Message Types 133
- B.1 Message Type 133
- B.2 Ordinary Messages 133
- B.3 High Priority Messages 133

Appendix C Utilities 135

Appendix D Debugging 137

Appendix E Configuration 139

Appendix F Administration 141
- F.1 Administrative Utilities 141
- F.2 System Controls 141
- F.3 /proc File System 141

Appendix G Examples 143
- G.1 Module Example 143
- G.2 Driver Example 143

Appendix H Copying 145
- H.1 GNU Affero General Public License 146
  - H.1.1 Preamble 146
  - H.1.2 How to Apply These Terms to Your New Programs 155
- H.2 GNU Free Documentation License 156

Glossary 163

Index 169

List of Figures

Figure 1.1: *Simple Stream* 16
Figure 1.2: *STREAMS-based Pipe* 17
Figure 1.3: *STREAMS-based FIFO (named pipe)* 17
Figure 1.4: *Stream to Communications Driver* 21
Figure 1.5: *A Message* 23
Figure 1.6: *Messages on a Message Queue* 25
Figure 1.7: *A Stream in More Detail* 27
Figure 1.8: *Many-to-one Multiplexor* 30
Figure 1.9: *One-to-many Multiplexor* 30
Figure 1.10: *Many-to-many Multiplexor* 30
Figure 1.11: *Internet Multiplexing Stream* 31
Figure 1.12: *Multiplexing Stream* 32
Figure 1.13: *Protocol Module Portability* 35
Figure 1.14: *Protocol Substitution* 36
Figure 1.15: *Protocol Migration* 37
Figure 1.16: *Module Reusability* 38
Figure 3.1: *Upstream and Downstream Stream Construction* 43
Figure 3.2: *Stream Queue Relationship* 44
Figure 3.3: *Opened STREAMS-based Driver* 46
Figure 3.4: *Opened STREAMS-based FIFO* 48
Figure 3.5: *Created STREAMS-based Pipe* 49
Figure 3.6: *Case Converter Module* 53
Figure 4.1: *Put Procedure Example* 58
Figure 5.1: *Message Form and Linkage* 66
Figure 5.2: *Data Buffer References* 67
Figure 5.3: *Message Ordering on a Queue* 74
Figure 5.4: *Message Ordering with One Priority Band* 75

List of Listings

Preface

Acknowledgements

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

Sponsors

Funding for completion of the *OpenSS7 OpenSS7* package was provided in part by:

- Monavacon Limited
- OpenSS7 Corporation

Additional funding for The OpenSS7 Project was provided by:

- Monavacon Limited
- AirNet Communications
- eServGlobal (NZ) Pty Ltd.
- France Telecom
- HOB International
- Motorola
- Newnet Communications
- Peformance Technologies
- SS8 Networks Inc.
- TECORE
- Verisign
- OpenSS7 Corporation
- Comverse Ltd.
- Excel Telecommunications
- GeoLink SA
- Lockheed Martin Co.
- NetCentrex S. A.
- Nortel Networks
- Sonus Networks Inc.
- SysMaster Corporation
- Tumsan Oy
- Vodare Ltd.

Contributors

The primary contributor to the *OpenSS7 OpenSS7* package is Brian F. G. Bidulock. The following is a list of notable contributors to The OpenSS7 Project:

- Per Berquist
- John Boyd
- Chuck Winters
- Peter Courtney
- Tom Chandler
- Gurol Ackman
- Pierre Crepieux
- Christopher Lydick
- Omer Tunali
- John Hodgkinson
- Kutluk Testicioglu
- John Wenker
- Angel Diaz
- Jérémy Compostella
- Sylvain Chouleur
- Christophe Nolibos
- Bryan Shupe
- D. Milanovic
- Tony Abo
- Others

Supporters

Over the years a number of organizations have provided continued support in the form of assessment, inspection, testing, validation and certification.

Telecommunications

- Integrated Telecom Solutions
- Accuris Networks
- Adax
- AASTRA
- Aculab
- AEPONA

- AirNet Communications
- Alacre
- Alcatel-Lucent
- Anam
- Alaska Power & Telephone
- Artesyn (now Emerson)
- Bharti Telesoft
- Continuous Computing (Trillium)
- Cisco
- Cogeco Cable Inc.
- Condor Networks
- Corecess
- Cosini
- Datacraft
- Datatronics
- Digium
- DTAG (Deutsche Telecom AG)
- Engage Communication Inc.
- eServGlobal (NZ) Pty Ltd.
- Excel Telecommunications
- France Telecom
- Geolink (now SeaMobile)
- Huawei
- Integral Access (now Telco Systems)
- Kineto Wireless
- Maestro Communications
- Mindspeed
- Mobixell
- Motorola
- m-Wise Inc.
- Net2Phone
- NetTest A/S (now Anritsu)
- Newnet Communications
- Noble Systems Corporation
- Nortel Networks
- OnMobile
- Ouroboros
- Primal Technologies Inc.
- Peformance Technologies
- Reliance Communications
- SONORYS Technology GmbH
- Spider Ltd. (now Emerson)
- Oasis Systems
- Stratus Technologies Bermuda Ltd.
- Switchlab Ltd.
- SysMaster Corporation
- Tecore
- Telcordia
- Teledesign
- Airwide Solutions
- Alcatel
- Altobridge
- Apertio (now Nokia)
- Aricent
- Arthus Technologies
- BubbleMotion
- Cellnext Solutions Limited
- Codent Networks
- Comverse Ltd.
- Coral Telecom
- Corelatus
- Data Connection
- Datatek Applications Inc.
- Dialogic
- Druid Software
- Empirix
- Ericsson
- ETSI
- Flextronics (now Aricent)
- Gemini Mobile Technologies
- Global Edge
- IBSYS Canada
- Integrat Mobile Aggregation Services
- Lucent
- MCI
- Mobis
- Motivity Telecom
- Mpathix Inc.
- Myriad Group
- NetCentrex S. A.
- NeuvaTel PCS
- NMS (now Dialogic)
- Nokia
- j2 Global Communications
- Orange
- P3 Solutions GmbH
- Propolys Pte Ltd.
- Pulse Voice Inc.
- Roamware Inc.
- Sonus Networks Inc.
- SS8 Networks Inc.
- Stratus
- Sicap AG
- Synapse Mobile Networks SA
- Tata Communications
- Tekno Telecom LLC
- Telecom Italia
- Telemetrics Inc.

- Telnor
- Texas Instruments Inc.
- Ulticom
- Vecto Communications SRL
- VeriSign
- VSE NET GmbH
- WINGcon GmbH
- Xentel Inc.
- ZTE Corporation
- TE-Systems
- Tumsan Oy
- Vanu Inc.
- Veraz Networks
- Vodare Ltd.
- The Software Group Limited
- Wipro Technologies
- YCOM SA

Aerospace and Military

- Advanced Technologies
- Altobridge
- ARINC
- ATOS Origin
- Boeing
- Boldon James
- CRNA
- DSNA-DGAC[1]
- DLR[2]
- DSNA-DTI
- Egis-Avia (Sofreavia)
- MetaSlash
- Sofreavia
- FAA WJHTC[3]
- Thales ATM/Air Systems
- Altobridge
- BBN (Bolt, Beranek, and Neuman)
- Boldon James
- Lockheed Martin Co.
- Northrop Grumman Corporation
- QinetiQ
- SAAB
- Sandia National Laboratories
- Thales
- Wright-Patterson Air Force Base

Financial, Business and Security

- Alebra
- Automated Trading Desk (now Citi)
- Banco Credicoop
- BeMac
- Boldon James
- CyberSource Corporation
- Fujitsu-Seimens
- FutureSoft
- Gcom
- GSX
- HOB International
- HP (Hewlett-Packard)
- IBM
- Lightbride (now CyberSource)
- MasterCard
- Network Executive Software Inc.
- Packetware Inc.
- Alebra
- Boldon James
- Fujitsu-Seimens
- FutureSoft
- GSX
- HOB International
- HP (Hewlett-Packard)
- IBM
- Alert Logic
- Apani
- BeMac

[1] La Direction des Services de la Navigation Aérienne - La Direction Général de l'Aviation Civile
[2] Deutsches Zentrum für Luft- unde Raumfarht
[3] Federal Aviation Administration - William J. Hughes Technical Center

- Packetware Inc.
- Prism Holdings Ltd.
- S2 Systems (now ACI)
- Symicron Computer Communications Limited
- ERCOM
- Hitech Systems
- iMETRIK
- Intrado Inc.

Education, Health Care and Nuclear Power

- IEEE Computer Society
- ENST[4]
- HTW-Saarland[5]
- Kansas State University
- University of North Carolina Charlotte
- Ateb
- Mandexin Systems Corporation
- Areva NP
- European Organization for Nuclear Research

Agencies

It would be difficult for the OpenSS7 Project to attain the conformance and certifications that it has without the free availability of specifications documents and standards from standards bodies and industry associations. In particular, the following:

- 3GPP (Third Generation Partnership Project)
- ATM Forum
- EIA/TIA (Electronic Industries Alliance)
- ETSI (European Telecommunications Standards Institute)
- ICAO (International Civil Aviation Organization)
- IEEE (Institute of Electrical and Electronic Engineers)
- IETF (The Internet Engineering Task Force)
- ISO (International Organization for Standardization)
- ITU (International Telecommunications Union)
- Mulutiservices Forum
- The Open Group

Of these, ICAO, ISO, IEEE and EIA have made at least some documents publicly available. ANSI is notably missing from the list: at one time draft documents were available from ANSI (ATIS), but that was curtailed some years ago. Telecordia does not release any standards publicly. Hopefully these organizations will see the light and realize, as the others have, that to remain current as a standards organization in today's digital economy requires providing individuals with free access to documents.

Authors

The authors of the *OpenSS7* package include:

– Brian Bidulock

Maintainer

The maintainer of the *OpenSS7* package is:

– Brian Bidulock

Please send bug reports to `bugs@openss7.org` using the `send-pr` script included in the package, only after reading the `BUGS` file in the release, or See ⟨undefined⟩ [⟨undefined⟩], page ⟨undefined⟩.

[4] Ecole Nationale Supérieure des Télécommunications

[5] Hochschule für Technik und Wirtschaft des Saarlandes

Document Information

Notice

This package is released and distributed under the *GNU Affero General Public License* (see Section H.1 [GNU Affero General Public License], page 146). Please note, however, that there are different licensing terms for the manual pages and some of the documentation (derived from OpenGroup[6] publications and other sources). Consult the permission notices contained in the documentation for more information.

This document, is released under the *GNU Free Documentation License* (see Section H.2 [GNU Free Documentation License], page 156) with no sections invariant.

Abstract

This document provides a *STREAMS Programmer's Guide* for *OpenSS7*.

Objective

The objective of this document is to provide a guide for the STREAMS programmer when developing STREAMS modules, drivers and application programs for *OpenSS7*.

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

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

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

Intent

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

Audience

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

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

Readers of the guide are expected to possess prior knowledge of the *Linux* and *UNIX* system, programming, networking, and data communication.

Revisions

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

A current version of this document is normally distributed with the *OpenSS7* package.

[6] Formerly X/Open and UNIX International.

Version Control

```
$Log: SPG2.texi,v $
Revision 1.1.2.3  2011-07-27 07:52:12  brian
- work to support Mageia/Mandriva compressed kernel modules and URPMI repo

Revision 1.1.2.2  2011-02-07 02:21:33  brian
- updated manuals

Revision 1.1.2.1  2009-06-21 10:40:06  brian
- added files to new distro
```

ISO 9000 Compliance

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

Disclaimer

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

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

U.S. Government Restricted Rights

If you are licensing this Software on behalf of the U.S. Government ("Government"), the following provisions apply to you. If the Software is supplied by the Department of Defense ("DoD"), it is classified as "Commercial Computer Software" under paragraph 252.227-7014 of the DoD Supplement to the Federal Acquisition Regulations ("DFARS") (or any successor regulations) and the Government is acquiring only the license rights granted herein (the license rights customarily provided to non-Government users). If the Software is supplied to any unit or agency of the Government other than DoD, it is classified as "Restricted Computer Software" and the Government's rights in the Software are defined in paragraph 52.227-19 of the Federal Acquisition Regulations ("FAR") (or any successor regulations) or, in the cases of NASA, in paragraph 18.52.227-86 of the NASA Supplement to the FAR (or any successor regulations).

Organization

This guide has several chapters, each discussing a unique topic. Chapter 1 [Introduction], page 13, Chapter 2 [Overview], page 39, Chapter 3 [Mechanism], page 41 and Chapter 4 [Processing], page 57 contain introductory information and can be ignored by those already familiar with STREAMS concepts and facilities.

This document is organized as follows:

[Preface], page 3

Describes the organization and purpose of the guide. It also defines an intended audience and an expected background of the users of the guide.

Chapter 1 [Introduction], page 13

An introduction to STREAMS and the *OpenSS7* package. STREAMS Fundamentals. Presents an overview and the benefits of STREAMS.

Chapter 2 [Overview], page 39

A brief overview of STREAMS.

Chapter 3 [Mechanism], page 41

A description of the STREAMS framework. Describes the basic operations for constructing, using, and dismantling Streams. These operations are performed using `open(2s)`, `close(2s)`, `read(2s)`, `write(2s)`, and `ioctl(2s)`.

Chapter 4 [Processing], page 57

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

Chapter 5 [Messages], page 61

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

Chapter 6 [Polling], page 87

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

Chapter 7 [Modules and Drivers], page 89

An overview of STREAMS modules, drivers and multiplexing drivers. Describes the STREAMS module and driver environment, input-output controls, routines, declarations, flush handling, driver-kernel interface, and also provides general design guidelines for modules and drivers.

Chapter 8 [Modules], page 91

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

Chapter 9 [Drivers], page 93

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

Chapter 10 [Multiplexing], page 95

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

Chapter 11 [Pipes and FIFOs], page 97

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

Chapter 12 [Terminal Subsystem], page 99

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

Chapter 13 [Synchronization], page 101
Discusses STREAMS in a symmetrical multi-processor environment.

Chapter 14 [Reference], page 111
Reference section.

Chapter 15 [Conformance], page 113
Conformance of the *OpenSS7* package to other *UNIX* implementations of STREAMS.

Chapter 16 [Portability], page 115
Portability of STREAMS modules and drivers written for other *UNIX* implementations of STREAMS and how they can most easily be ported into *OpenSS7*; but, for more details on this topic, see the *OpenSS7 - STREAMS Portability Guide*.

Chapter 17 [Developing Portable STREAMS Modules], page 127
Development guidelines for developing portable STREAMS modules and drivers.

Appendix A [Data Structures], page 131
Primary STREAMS Data Structures, descriptions of their members, flags, constants and use. Summarizes data structures commonly used by STREAMS modules and drivers.

Appendix B [Message Types], page 133
STREAMS Message Type reference, with descriptions of each message type. Describes STREAMS messages and their use.

Appendix C [Utilities], page 135
STREAMS kernel-level utility functions for the module or driver writer. Describes STREAMS utility routines and their usage.

Appendix D [Debugging], page 137
STREAMS debugging facilities and their use. Provides debugging aids for developers.

Appendix E [Configuration], page 139
STREAMS configuration, the *STREAMS Administrative Driver* and the autopush facility. Describes how modules and drivers are configured into the *Linux* and *UNIX* system, tunable parameters, and the autopush facility.

Appendix F [Administration], page 141
Administration of the STREAMS subsystem.

Appendix G [Examples], page 143
Collected examples.

Conventions Used

This guide uses *texinfo* typographical conventions.

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

Examples are given to highlight the most important and common capabilities of STREAMS. They are not exhaustive and, for simplicity, often reference fictional drivers and modules. Some examples are also present in the *OpenSS7* package, both for testing and example purposes.

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

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

Declarations and short examples are in `texinfo` '`sample`' typesetting.

`texinfo` displays are used to show program source code.

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

Other Documentation

Although the *STREAMS Programmer's Guide* for *OpenSS7* provides a guide to aid in developing STREAMS applications, readers are encouraged to consult the *OpenSS7* manual pages. For a reference for writing code, the manual pages (see `STREAMS(9)`) provide a better reference to the programmer. For detailed information on system calls used by STREAMS (section 2), and STREAMS utilities from section 8. STREAMS specific input output control (ioctl) calls are provided in `streamio(7)`. STREAMS modules and drivers are described on section 7. STREAMS is also described to some extent in the *System V Interface Definition, Third Edition*.

UNIX Edition

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

Related Manuals

OpenSS7 Installation and Reference Manual

Copyright

© 1997-2014 Monavacon Limited. All Rights Reserved.

1 Introduction

1.1 Background

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

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

On *UNIX System V Relase 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 FIFOs[2] or terminal input-output.[3].

Linux has not traditionally implemented a STREAMS subsystem. It is not clear why, however, perceived ideological differences between STREAMS and *Sockets* and also the *XTI/TLI* and *Sockets* interfaces to *Internet Protocol* services are usually at the centre of the debate. For additional details on the debate, see Section "About This Manual" in *OpenSS7 Frequently Asked Questions*.

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

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

OpenSS7 is the current open source implementation of STREAMS for *Linux* and provides all of the capabilities of *UNIX System V Release 4.2 MP*, plus support for mainstream *UNIX* implementations based on *UNIX System V Release 4.2 MP* through compatibility modules.

Although it is intended primarily as documentation for the *OpenSS7* implementation of STREAMS, much of the *OpenSS7 - STREAMS Programmer's Guide* is generally applicable to all STREAMS implementations.

---

[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 What is STREAMS?

STREAMS is a flexible, message oriented framework for the development of *GNU/Linux* communications facilities and protocols. It provide a set of system calls, kernel resources, and kernel utilities within a framework that is applicable to a wide range of communications facilities including terminal subsystems, interprocess communication, and networking. It provides standard interfaces for communication input and output within the kernel, common facilities for device drivers, and a standard interface[4] between the kernel and the rest of the *GNU/Linux* system.

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

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

1.2.1 Characteristics

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

- STREAMS is based on the character device special file which is one of the most flexible special files available in the *GNU/Linux* system.
- STREAMS is a message passing architecture, similar to *Message Oriented Middleware (MOM)* that achieves a high degree of functional decoupling between modules. This allows the service interface between modules to correspond to the natural interfaces found or described between protocol layers in protocol stack without requiring the implementation to conform to any given model.

  As a contrasting example, the *BSD Sockets* implementation, internal to the kernel, provides strict socket-protocol, protocol-protocol and protocol-device function call interfaces.
- By using `put` and `service` procedures for each module, and scheduling `service` procedures, STREAMS combines background scheduling of coroutine service procedures with message queueing and flow control to provide a mechanism robust for both event driven subsystem and soft real-time subsystem.

  In contrast, *BSD Sockets*, internal to the kernel, requires the sending component across the socket-protocol, protocol-protocol, or protocol-device to handle flow control. STREAMS integrates flow control within the STREAMS framework.
- STREAMS permits user runtime configuration of kernel data structure and modules to provide for a wide range of novel configurations and capabilities in a live *GNU/Linux* system. The *BSD Sockets* protocol framework does not provide this capability.
- STREAMS is as applicable to termination input-output and interprocess communication as it is to networking protocols.

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

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

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

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

1.2.2 Components

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

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

- A *Stream head* that is inside the *Linux* kernel, but which sits closest to the user space process. The *Stream head* is responsible for communicating with user space processes and that presents the standard STREAMS I/O interface to user space processes and applications.

- A *Stream end* or *Driver* that is inside the *Linux* kernel, but which sits farthest from the user space process. A *Stream end* or *Driver* that interfaces to hardware or other mechanisms within the *Linux* kernel.

- A *Module* that sits between the *Stream head* and *Stream end*. The *Module* provides modular and flexible processing of control and data information passed up and down the *Stream*.

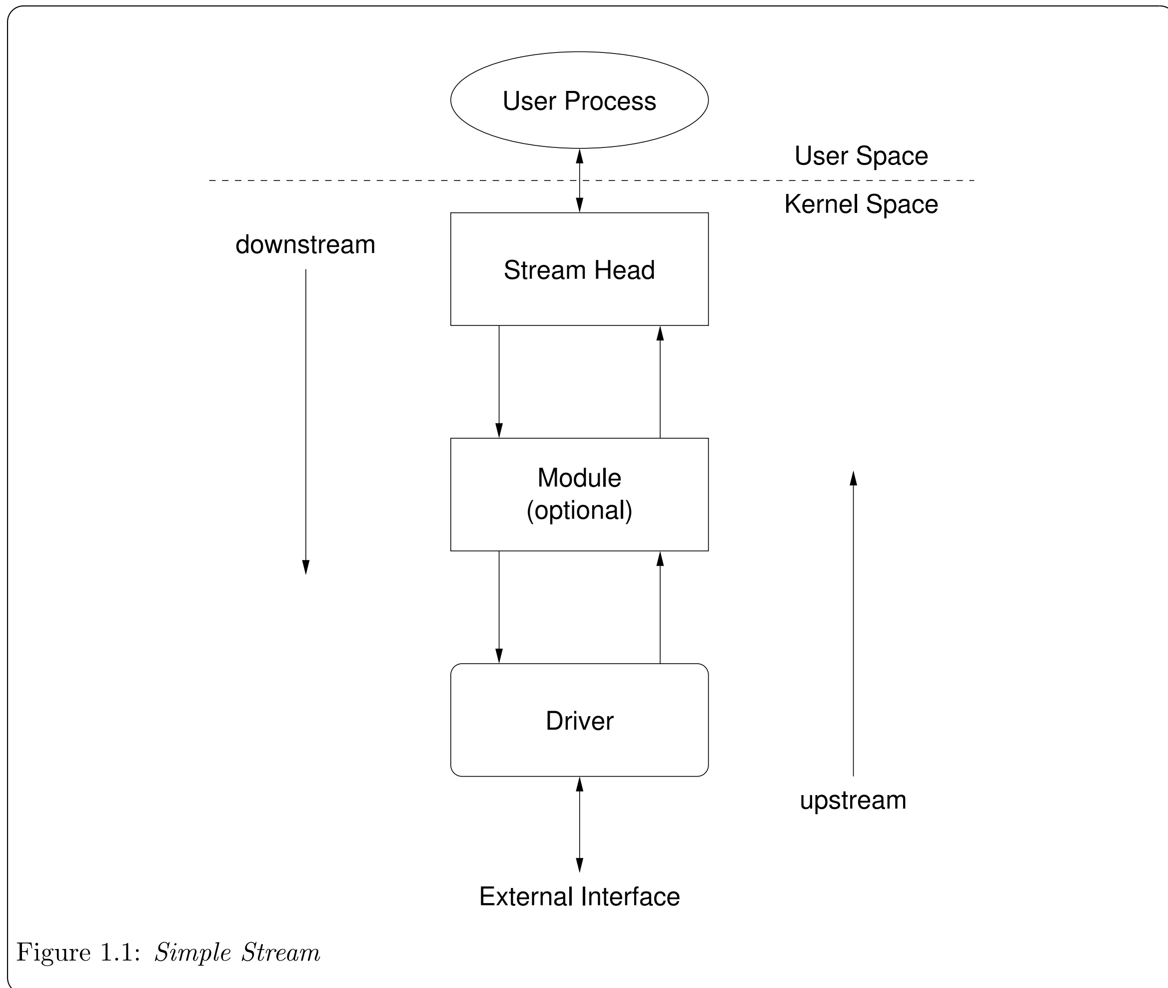

Figure 1.1: *Simple Stream*

1.2.2.1 Stream head

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

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

STREAMS-based pipes place another *Stream head* in the position of the *Driver*. That is, a STREAMS-based pipe is a full-duplex communications path between two otherwise independent *Stream heads*. *Modules* may be placed between the *Stream heads* in the same fashion as they can exist between a *Stream head* and a *Driver* in a normal *Stream*. A STREAMS-based pipe is illustrated in Figure 1.2.

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

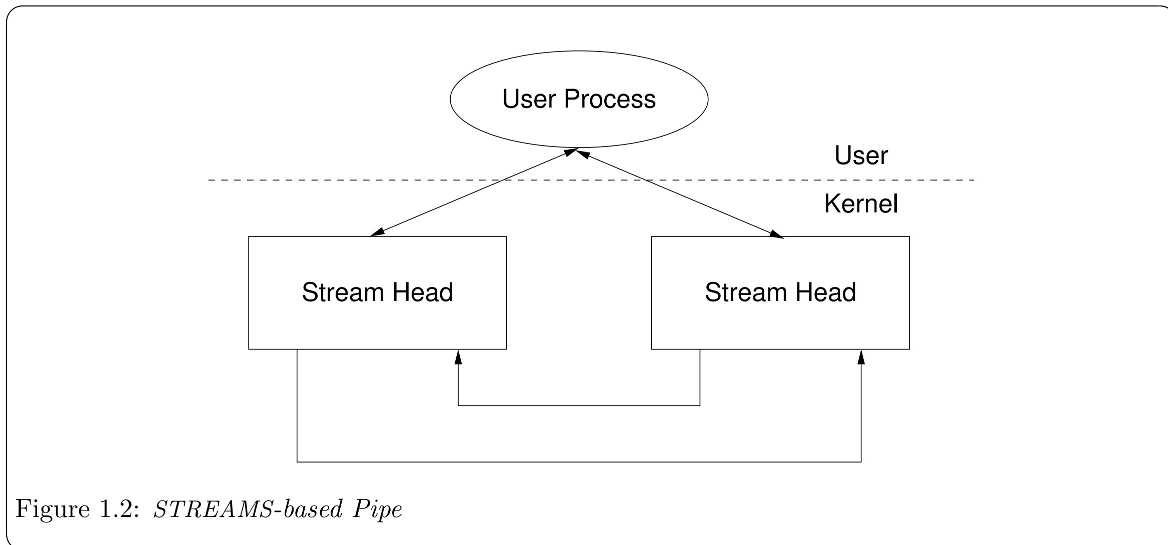

Figure 1.2: *STREAMS-based Pipe*

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

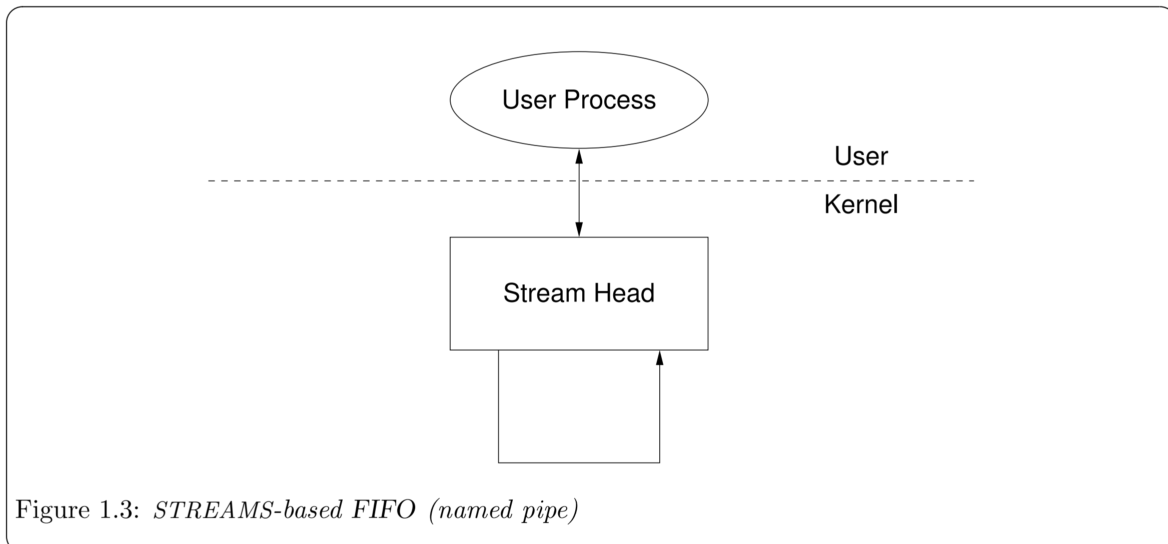

Figure 1.3: *STREAMS-based FIFO (named pipe)*

For more information on STREAMS-based pipes and FIFOs, see Chapter 11 [Pipes and FIFOs], page 97.

1.2.2.2 Module

A STREAMS *Module* is an optional processing element that is placed between the *Stream head* and the *Stream end*. The *Module* can perform processing functions on the data and control information flowing in either direction on the *Stream*. It can communicate with neighbouring modules, the *Stream head* or a *Driver* using STREAMS messages. Each *Module* is self-contained in the sense that it does not directly invoke functions provided by, nor access data structures of, neighbouring modules, but rather communicates data, status and control information using messages. This functional

isolation provides a loose coupling that permits flexible recombination and reuse of *Modules*. A *Module* follows the same framework as the *Stream head* and *Driver*, has all of the same entry points and can use all of the same STREAMS and kernel utilities to perform its function.

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

For more information on STREAMS modules, see Section 1.4.3 [Modules], page 26.

1.2.2.3 Driver

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

For more information on STREAMS drivers, see Section 1.4.4 [Drivers], page 28.

1.2.2.4 Queues

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

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

For more information on STREAMS queues, see Section 1.4.1 [Queues], page 21.

1.2.2.5 Messages

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

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

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

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

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

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

For more information on STREAMS messages, see Section 1.4.2 [Messages], page 22.

1.3 Basic Streams Operations

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

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

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

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

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

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

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

1.3.1 Basic Operations Example

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

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

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

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

Listing 1.1: *Basic Operations*

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

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

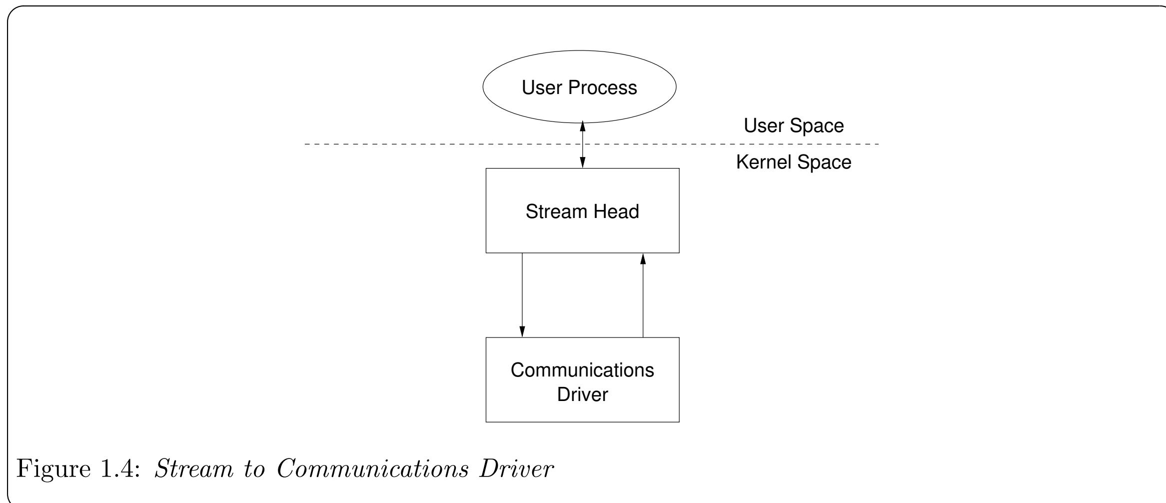

Figure 1.4: *Stream to Communications Driver*

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

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

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

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

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

1.4 Components

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

1.4.1 Queues

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

A *queue* provides an interface between an instance of a STREAMS driver, module or *Stream head*, and the other modules and drivers that make up a *Stream* for a direction of message flow (i.e., *upstream* or *downstream*). When an instance of a STREAMS driver, module or *Stream head* is associated with a *Stream*, a pair of queues are allocated to represent the driver, module or *Stream head* within the *Stream*. Queue data structures are always allocated in pairs. The first queue in the pair is the *read-side* or *upstream* queue in the pair; the second queue, the *write-side* or *downstream* queue.

Queues are described in greater detail in Section 5.3 [Queues and Priority], page 74.

1.4.1.1 Queue Procedures

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

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

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

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

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

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

The queue `close` procedure for a module is called whenever the module is popped (removed) from a *Stream*. Modules are automatically popped from a *Stream* on the last close of the *Stream*. The queue `close` procedure for a driver is called with the last close of the *Stream* or when the last reference to the *Stream* is relinquished. If the *Stream* is linked under a multiplexing driver (`I_LINK(7)`), or has been named with `fattach(3)`, then the *Stream* will not be dismantled on the last close and the `close` procedure not called until the *Stream* is eventually unlinked (`I_UNLINK(7)`) or detached (`fdetach(3)`).

Procedures are described in greater detail in Section 4.1 [Procedures], page 57.

1.4.2 Messages

This subsection provides a brief overview of STREAMS messages.

In fitting with the concept of function decoupling, all control and data information is passed between STREAMS modules, drivers and the *Stream head* using messages. Utilities are provided to the STREAMS module writer for passing messages using queue and message pointers. STREAMS messages consist of a 3-tuple of a message block structure (`msgb(9)`), a data block structure (`datab(9)`) and a data buffer. The message block structure is used to provide an instance of a reference to a data block and pointers into the data buffer. The data block structure is used to provide information about the data buffer, such as message type, separate from the data contained in the buffer. Mes-

sages are normally passed between STREAMS modules, drivers and the *Stream head* using utilities that invoke the target module's `put` procedure, such as `put(9s)`, `putnext(9)`, `qreply(9)`. Messages travel along a *Stream* with successive invocations of each driver, module and *Stream head*'s `put` procedure.

Messages are described in greater detail in Section 5.1 [Messages Overview], page 61 and Chapter 5 [Messages], page 61.

1.4.2.1 Message Types

This subsection provides a brief overview of STREAMS message types.

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

Most of the defined message types (see Section 5.1.1 [Message Types], page 61, and Appendix B [Message Types], page 133) are solely for use within the STREAMS framework. A more limited set of message types (`M_PROTO`, `M_PCPROTO` and `M_DATA`) can be used to pass control and data information to and from the user process via the *Stream head*. These message type can be generated and consumed using the `read(2s)`, `write(2s)`, `getmsg(2s)`, `getpmsg(2s)`, `putmsg(2s)`, `putpmsg(2s)` system calls and some `streamio(7)` STREAMS `ioctl(2s)`.

Message types are described in detail in Section 5.1.1 [Message Types], page 61 and Appendix B [Message Types], page 133.

1.4.2.2 Message Linkage

Messages blocks of differing types can be linked together into composite messages as illustrated in Figure 1.5.

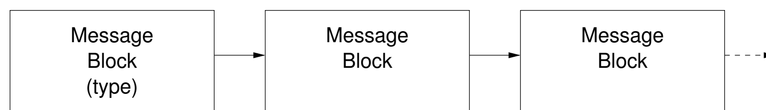

Figure 1.5: *A Message*

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

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

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

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

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

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

Two messages linked together on a message queue is illustrated in Figure 1.6. In the figure, '`Message 2`' is linked to '`Message 1`'.

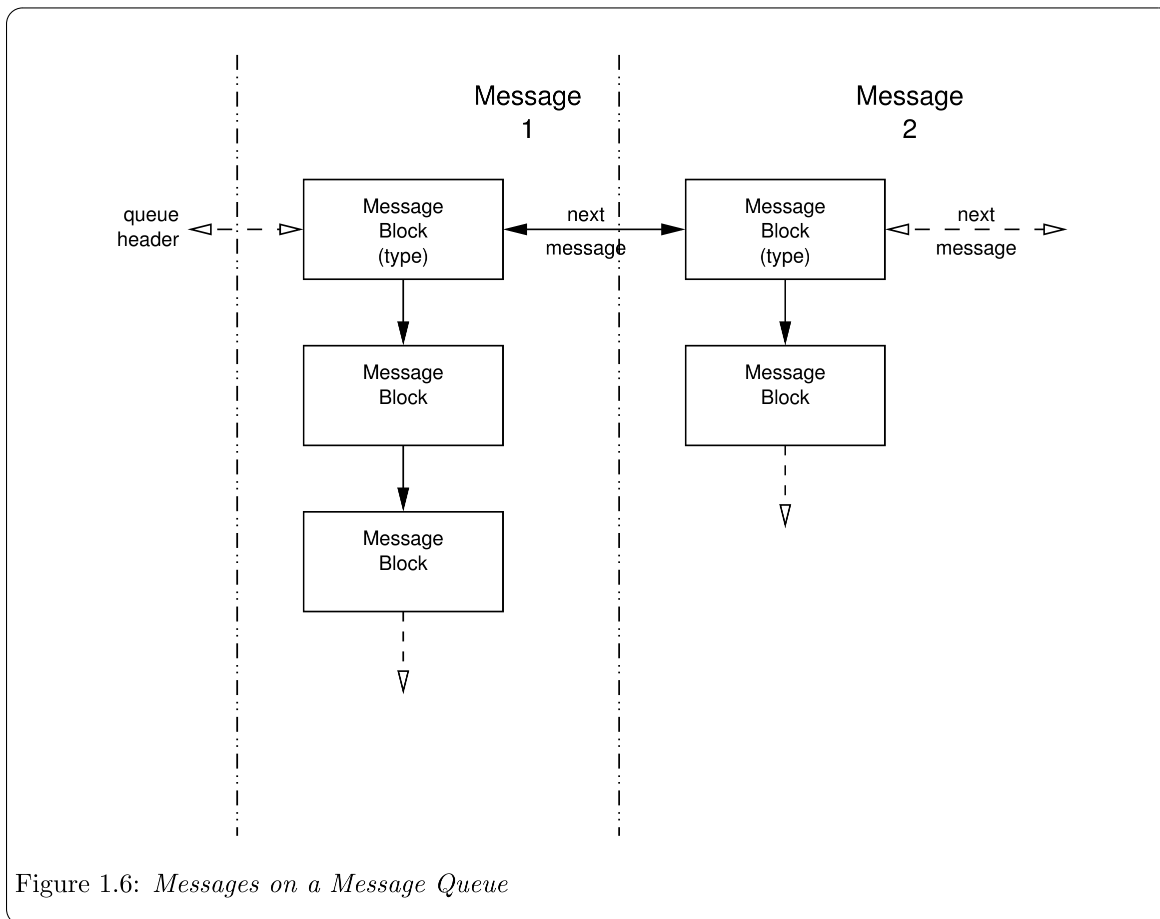

Figure 1.6: *Messages on a Message Queue*

As illustrated in Figure 1.6, when a message exists on a message queue, the first message block in the message (which can possibly contain a chain of message blocks) is linked into a double linked list used by the message queue to order and track messages. The queue structure, `queue(9)`, contains the head and tail pointers for the linked list of messages that reside on the queue. Some of the fields in the first message block (such as the linked list pointers) are significant only in the first message block of the message and applies to all the message blocks in the message (such as message band). Message linkage is described in detail in Section 5.2 [Message Structure], page 63.

1.4.2.3 Message Queueing Priority

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

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

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

Normal messages are queued in priority band '`0`'. Priority messages are queued in bands greater than zero ('`1`' through '`255`' inclusive). Messages of a higher ordinal band number are of greater priority. For example, a priority message for band '`23`' is queued ahead of messages for band '`22`'. Normal and priority messages are subject to flow control within a *Stream*, and a queued according to priority.

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

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

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

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

Queueing priority is described in greater detail in Section 5.3 [Queues and Priority], page 74.

1.4.3 Modules

This subsection provides a brief overview of STREAMS modules.

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

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

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

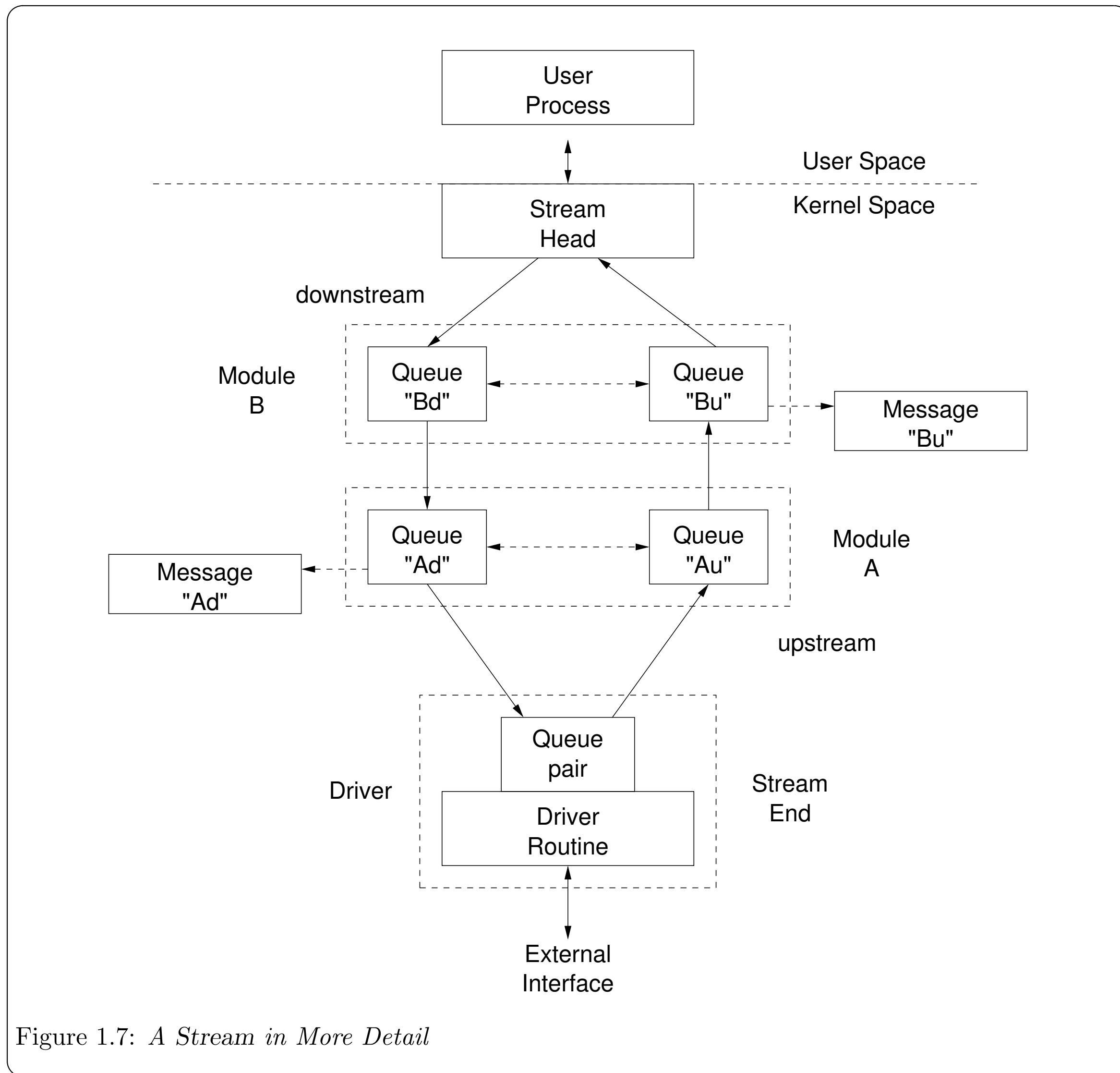

Figure 1.7: *A Stream in More Detail*

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

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

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

Messages

Messages can be inserted into, and removed from, the linked list message queue associated with each queue in the queue pair as they pass through the module. For example, in Figure 1.7, '`Message Ad`' exists on the '`Ad`' queue; '`Message Bu`', on the '`Bu`' queue.

Processing Procedures

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

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

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

Data

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

Modules are described in greater detail in Chapter 8 [Modules], page 91.

1.4.4 Drivers

This subsection provides a brief overview of STREAMS drivers.

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

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

1. Drivers are responsible for generating and consuming messages at the *Stream end*.

   Drivers convert STREAMS messages into appropriate software or hardware actions, events and data transfer. As a result, drivers that are associated with a hardware device normally contain an interrupt service procedure that handles the external device specific actions, events and data transfer. Messages are typically consumed at the *Stream end* in the driver's downstream `put` or `service` procedure and action take or data transferred to the hardware device. Messages are typically generated at the *Stream end* in the driver's interrupt service procedure, and inserted upstream using the `put(9s)` STREAMS utility.

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

   In contrast, *modules* are intended to operate solely within the STREAMS framework.
2. Because a driver sits at a *Stream* end and can support multiplexing, a driver can have multiple *Streams* connected to it, either upstream (fan-in) or downstream (fan-out) (see Section 1.5 [Multiplexing], page 29).

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

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

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

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

Drivers are described in greater detail in Chapter 9 [Drivers], page 93.

1.4.5 Stream Head

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

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

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

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

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

Stream heads are described in greater detail in Chapter 3 [Mechanism], page 41, Chapter 6 [Polling], page 87, Chapter 11 [Pipes and FIFOs], page 97, and Chapter 12 [Terminal Subsystem], page 99.

1.5 Multiplexing

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

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

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

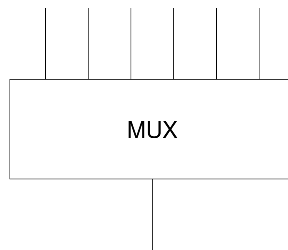

Figure 1.8: *Many-to-one Multiplexor*

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

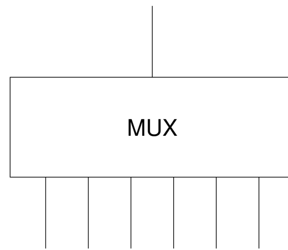

Figure 1.9: *One-to-many Multiplexor*

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

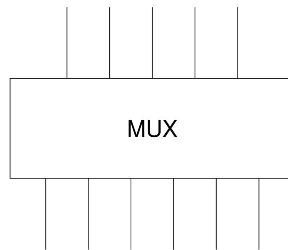

Figure 1.10: *Many-to-many Multiplexor*

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

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

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

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

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

1.5.1 Fan-Out Multiplexers

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

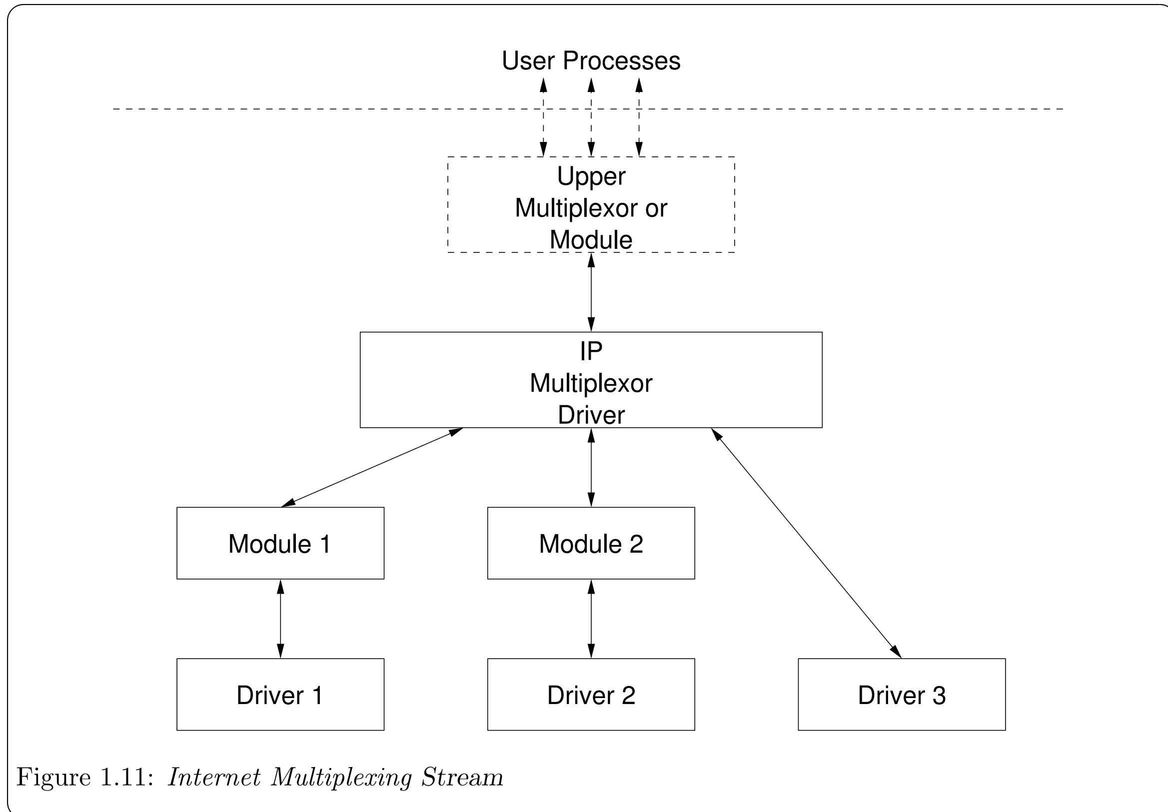

Figure 1.11: *Internet Multiplexing Stream*

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

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

1.5.2 Fan-In Multiplexers

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

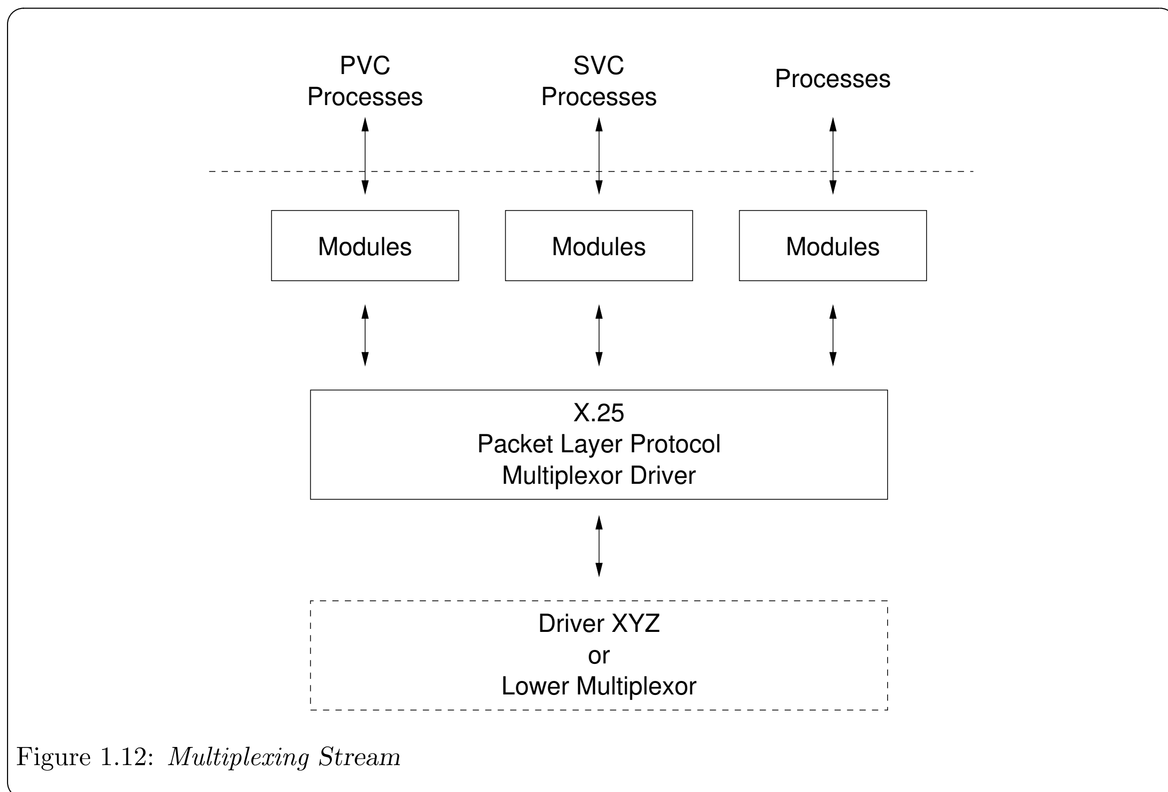

Figure 1.12: *Multiplexing Stream*

The ability to multiplex upper *Streams* to a driver is a characteristic supported by all STREAMS drivers: not just *multiplexing drivers*. Each `open(2s)` to a minor device node results in another upper *Stream* that can be associated with the device driver. What the *multiplexing driver* permits

over the normal STREAMS driver is the ability to link one or more lower *Streams* (possibly containing modules and another multiplexing driver) beneath it.

1.5.3 Complex Multiplexers

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

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

Multiplexing is described in greater detail in Chapter 10 [Multiplexing], page 95.

1.6 Benefits of STREAMS

STREAMS provides a flexible, scalable, portable, and reusable kernel and user level facility for the development of *GNU/Linux* system communications services. STREAMS allows the creation of kernel resident modules that offer standard message passing facilities and the ability for user level processes to manipulate and configure those modules into complex topologies. STREAMS offers a standard way for user level processes to select and interconnect STREAMS modules and drivers in a wide array of combinations without the need to alter *Linux* kernel code, recompile or relink the kernel.

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

1.6.1 Standardized Service Interfaces

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

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

A service interface is defined[7] at the boundary between neighbouring modules. The upper module at the boundary is termed the *service user* and the lower module at the boundary is termed the *service provider*. Implemented under STREAMS, a service interface is a specified set of messages and the rules that allow passage of these messages across the boundary. A STREAMS module or driver that implements a service interface will exchange messages within the defined set across the boundary and will respond to received messages in accordance with the actions defined for the specific message

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

and the sequence of messages preceding receipt of the message (i.e., in accordance with the state of the module).

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

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

1.6.2 Manipulating Modules

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

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

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

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

1.6.2.1 Protocol Portability

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

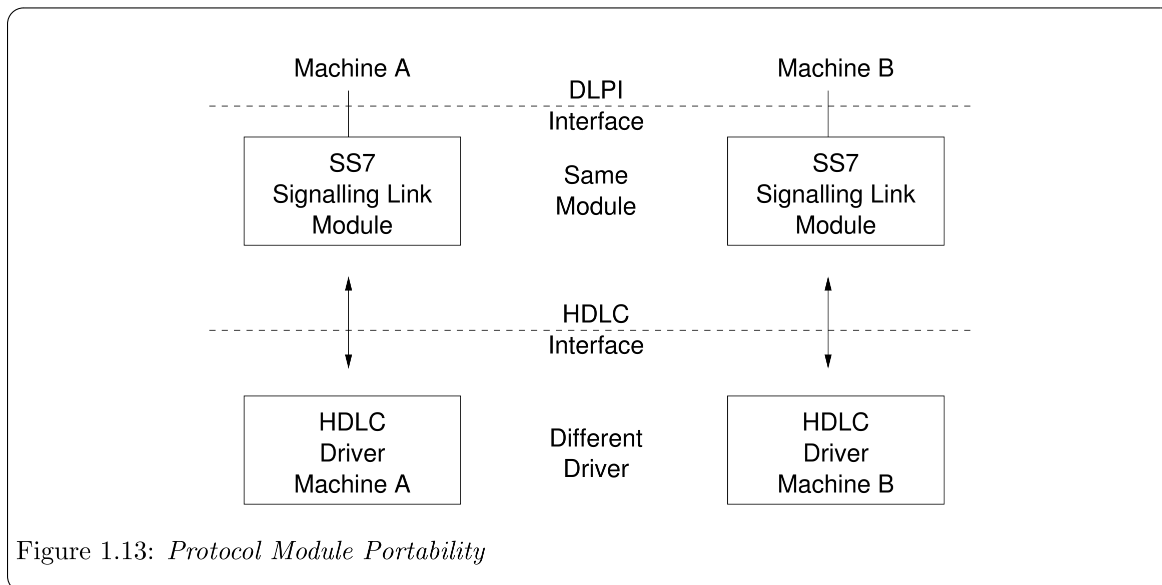

Figure 1.13: *Protocol Module Portability*

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

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

With *OpenSS7*, STREAMS provides the ability to directly port a large body of existing STREAMS modules to the *GNU/Linux* operating system.

1.6.2.2 Protocol Substitution

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

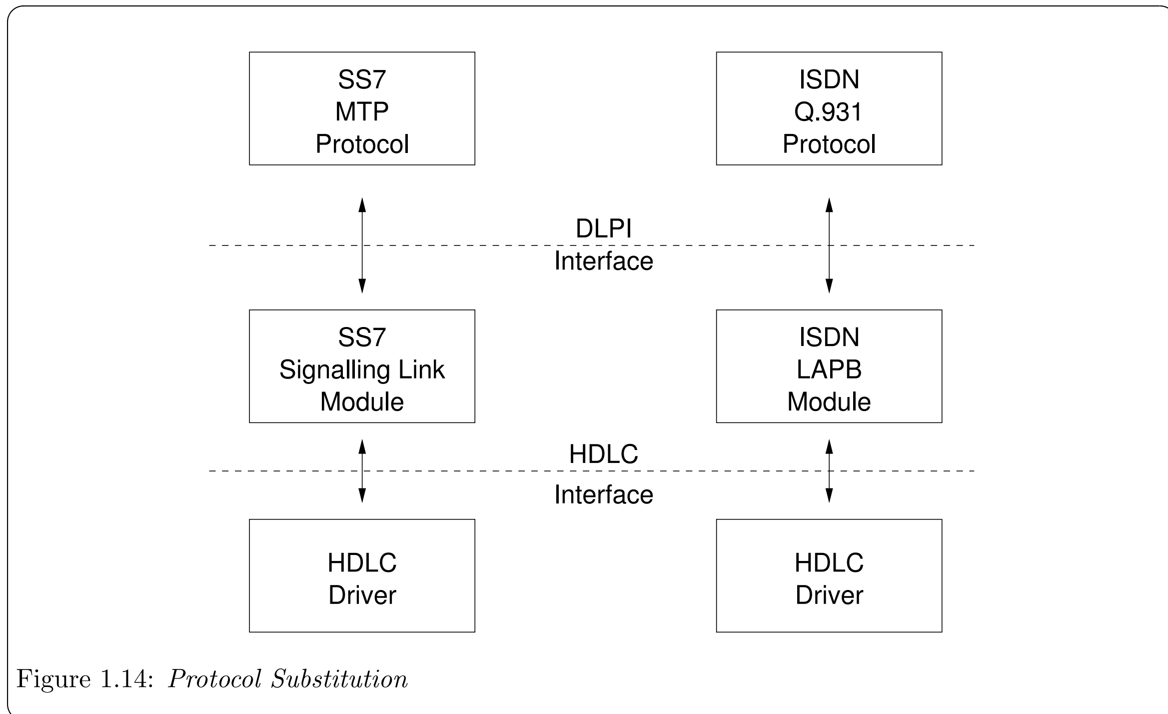

Figure 1.14: *Protocol Substitution*

Figure 1.14 illustrates how STREAMS can substitute upper layer protocol modules to implement a different protocol stack over the same *HDLC* driver. As each module and driver support the same service interface at each level, it is conceivable that the resulting modules could be recombined to support, for example, *SS7 MTP* over an *ISDN LAPB* channel.[8]

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

1.6.2.3 Protocol Migration

Figure 1.15 illustrates how STREAMS can move functions between kernel software and front end firmware. A common downstream service interface allows the transport protocol module to be independent of the number or type of modules below. The same transport module will connect without modification to either an *SS7 Signalling Link* module or *SS7 Signalling Link* driver that presents the same service interface.

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

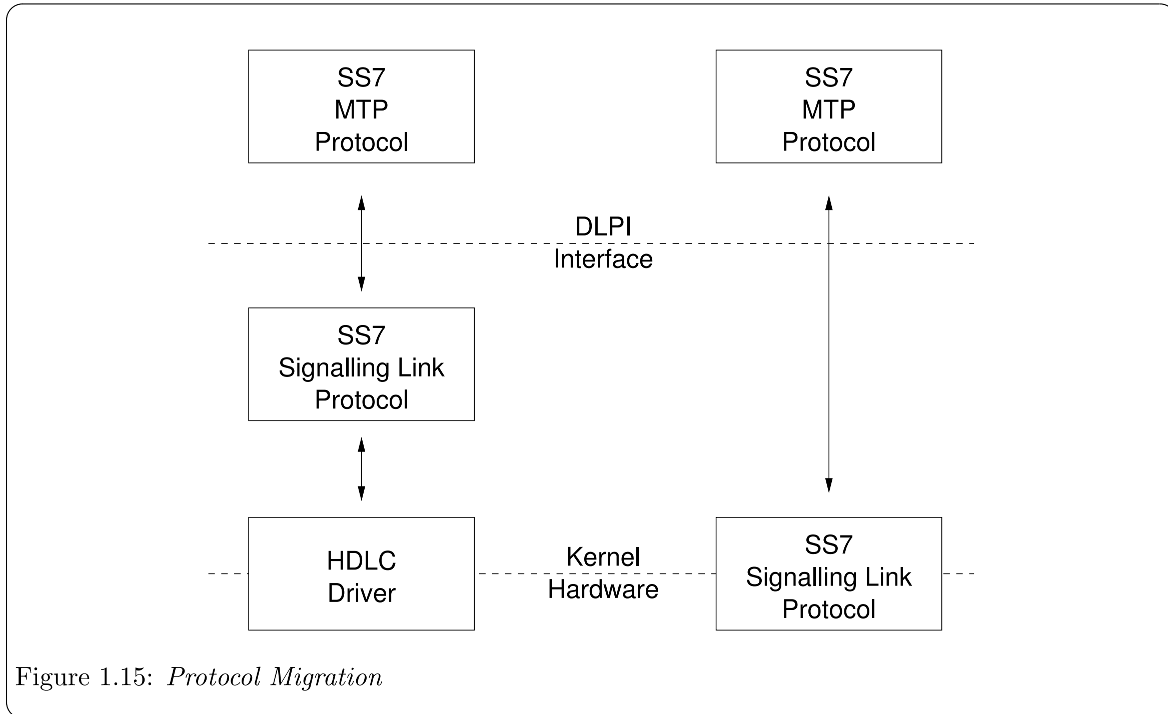

Figure 1.15: *Protocol Migration*

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

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

1.6.2.4 Module Reusability

Figure 1.16 shows the same canonical module (for example, one that provides delete and kill processing on character strings) reused in two different *Streams*. This module would typically be implemented as a filter, with no downstream service interface. In both cases, a tty interface is presented to the *Stream*'s user process since the module is nearest the *Stream head*.

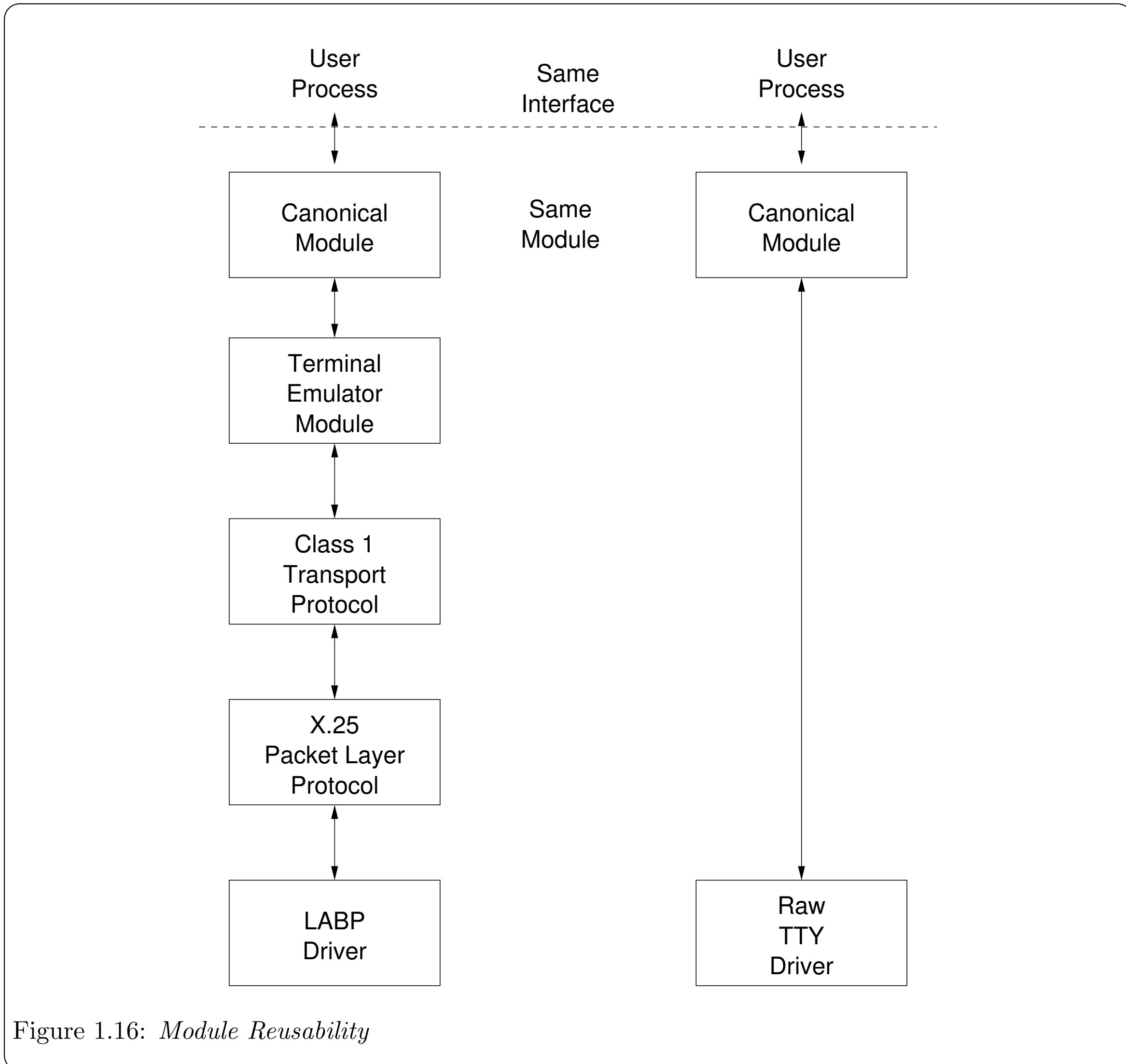

Figure 1.16: *Module Reusability*

2 Overview

2.1 Definitions

2.2 Concepts

2.3 Application Interface

2.4 Kernel Level Facilities

2.5 Subsystems

3 Mechanism

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

3.1 Mechanism Overview

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

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

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

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

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

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

The `poll(2s)` system call[1] provides the ability for the application to poll multiple *Streams* for a wide range of events.

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

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

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

STREAMS also provides kernel level utilities and facilities for the development of kernel resident STREAMS modules and drivers. Within the STREAMS framework, the *Stream head* is responsible for conversion between STREAMS messages passed up and down a *Stream* and the system call interface presented to user level applications programs. The *Stream head* is common to all STREAMS special

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

files and the conversion between the system call interface and message passed on the *Stream* does not have to be reimplemented by the module and device driver writer as is the case for traditional character device I/O.

3.1.1 STREAMS System Calls

The STREAMS-related system calls are:

`open(2s)` Open a STREAMS special file and create a new (or access an existing) *Stream*.

`close(2s)` Close a STREAMS special file and possibly cause the destruction of a *Stream* (i.e., on the last close of the *Stream*.

`read(2s)` Read data from an open *Stream*.

`write(2s)` Write data to an open *Stream*.

`ioctl(2s)` Control an open *Stream*.

`getmsg(2s)`, `getpmsg(2s)` Receive a (prioritized) message at the *Stream head*.

`putmsg(2s)`, `putpmsg(2s)` Send a (prioritized) message from the *Stream head*.

`poll(2s)` Receive notification when selected events occur on one or more *Streams*.

`pipe(2s)` Create a channel that provides a STREAMS-based bidirectional communication path between multiple processes.

3.2 Stream Construction

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

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

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

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

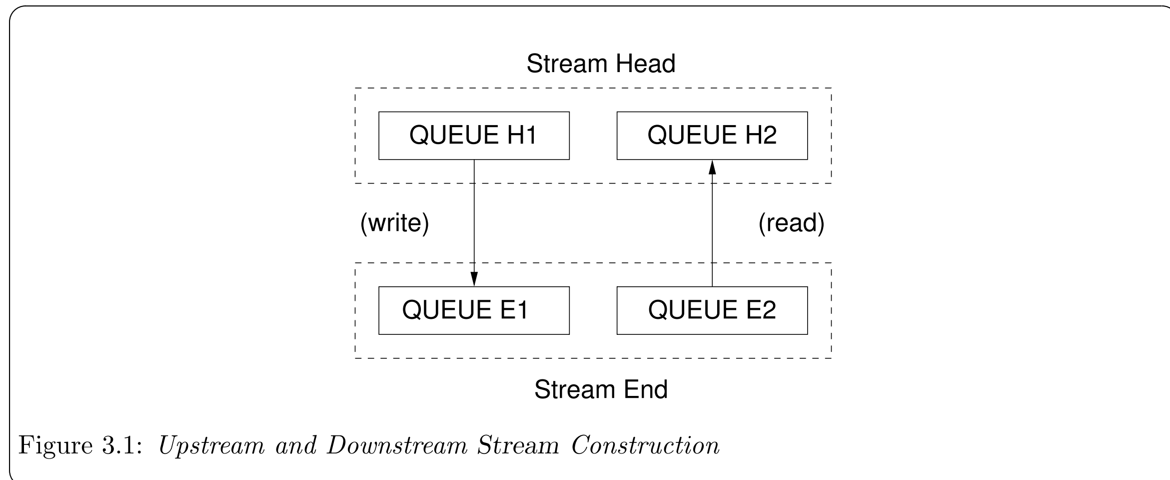

Figure 3.1: *Upstream and Downstream Stream Construction*

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

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

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

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

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

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

The `module_stat(9)` structure contains runtime counts of the entry into the various procedures contained in the *qinit(9)* structure as well as a pointer to any module private statistics that need to be collected. As depicted in Figure 3.2, there is normally only one `module_stat(9)` structure

per queue pair that collects statistics for the entire module or driver. STREAMS does not peg this counts automatically and will not manipulate this structure, even when one is attached. It is the responsibility of the module or driver writer to peg counts as required. *OpenSS7* does, however, provide some user level administrative tools that can be used to examine the statistics contained in this structure. The `module_stat(9)` structure is opaque to the STREAMS subsystem and can be read from or written to by module or driver procedures.

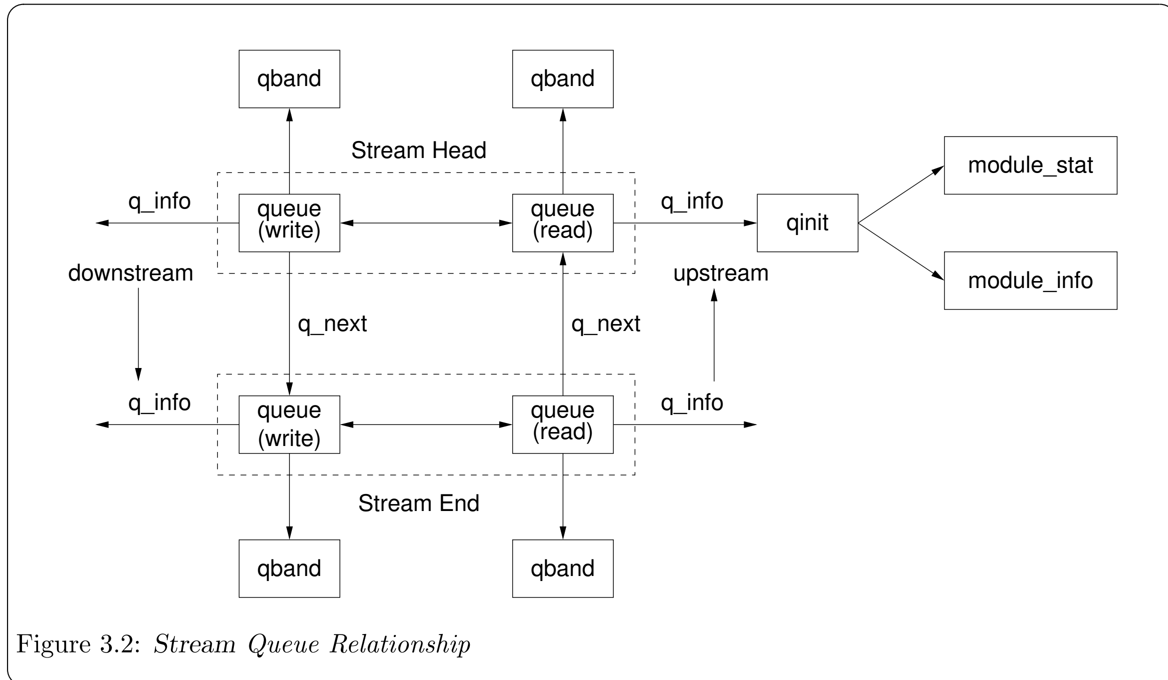

Figure 3.2: *Stream Queue Relationship*

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

All of these queue related data structures are in Appendix A [Data Structures], page 131 (and in the *OpenSS7 Manual Pages*).

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

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

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

1. Open a STREAMS device special file using the `open(2s)` system call. Construction of a *Stream* with the *open(2s)* system call is detailed in Section 3.2.1 [Opening a STREAMS Device File], page 45 and Section 3.2.2 [Opening a STREAMS-based FIFO], page 47 and illustrated in Figure 3.3.

2. Create a STREAMS-based pipe using the `pipe(2s)` system call. Construction of a *Stream* with the *pipe(2s)* system call is detailed in Section 3.2.3 [Creating a STREAMS-based Pipe], page 49 and illustrated in Figure 3.5.

3.2.1 Opening a STREAMS Device File

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

In the traditional *UNIX* system, a STREAMS-based driver file is a character device special file within the *UNIX* file system. In the *GNU/Linux* system, under *OpenSS7*, a STREAMS-based driver file is either a character device special file within a *GNU/Linux* file system, or a character device special file within the mounted *Shadow Special File System (specfs)*. When the `specfs` is mounted, `specfs` device nodes can be opened directly. When the `specfs` is not mounted, `specfs` device nodes can only be opened indirectly via character device nodes in a *GNU/Linux* file system external to the `specfs`.

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

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

The `streamtab` structure defines a module or driver. *st_rdinit* points to the read `qinit` structure for the driver and *st_wrinit* points to the driver's write `qinit` structure. For a multiplexing driver, the *st_muxrinit* and *st_muxwinit* point to the `qinit` structures for the lower side of the multiplexing driver. For a regular non-multiplexing driver these members are `NULL`.

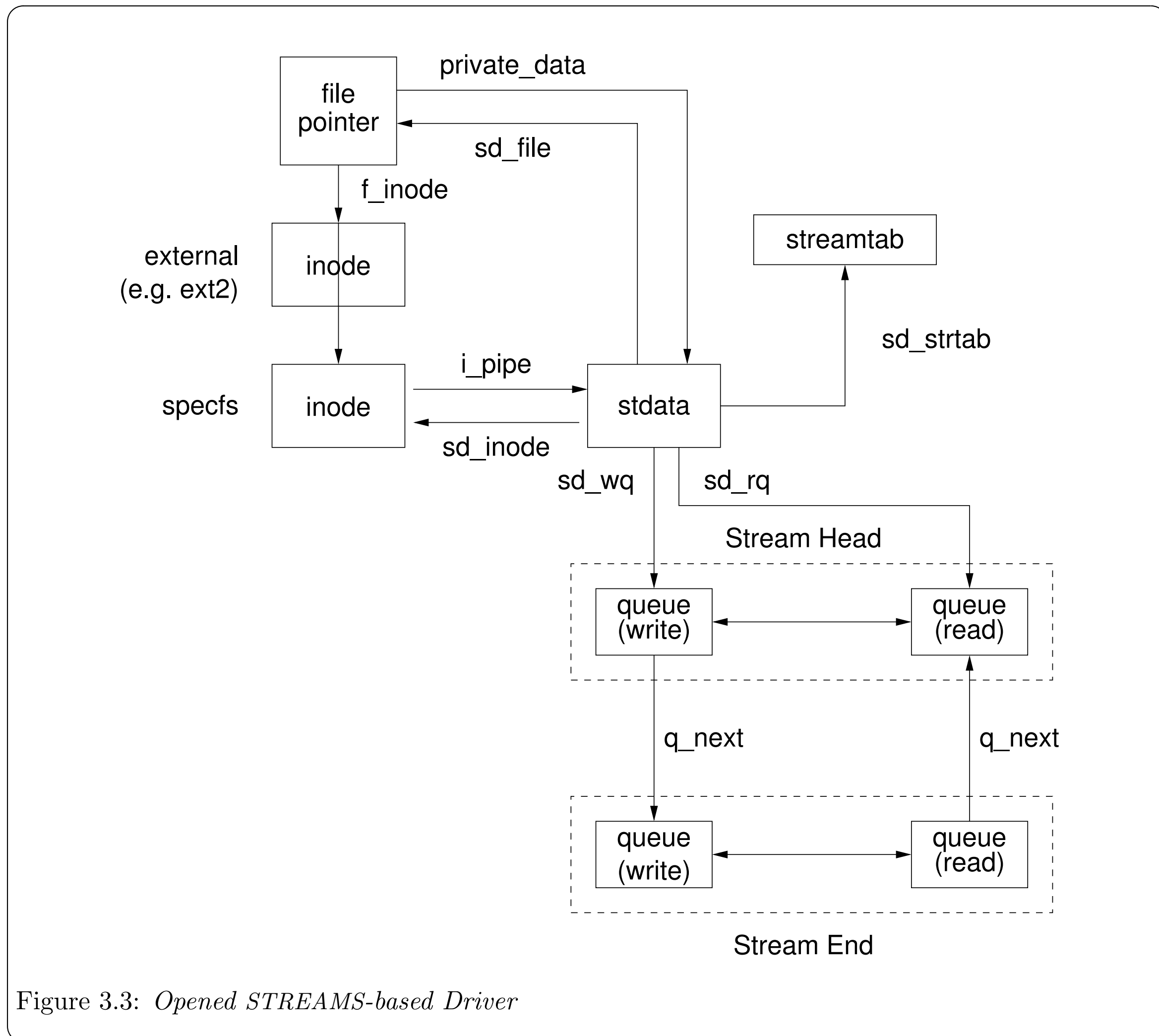

Figure 3.3: *Opened STREAMS-based Driver*

3.2.1.1 First Open of a Stream

When a STREAMS-based file is opened, a new *Stream* is created if one does not already exists for the file, or if the `D_CLONE` flag is set for the file indicating that a new *Stream* is to be created on each open of the file. First, a file descriptor is allocated in the process' file descriptor table, a file pointer is allocated to represent the opened file. The file pointer is initialized to point to the `inode` associated with the character special file in the external file system (see *f_inode* in Figure 3.3). This `inode` is of type character special (`S_IFCHR`). The *Linux* kernel recognizes the `inode` as a character special file and invokes the character device open routine in *OpenSS7*. This `inode` is equivalent to the `vnode` used by *UNIX System V Release 4.2*.

OpenSS7 uses the major and minor device numbers associated with the character special file to locate an `inode` within the *Shadow Special File System (specfs)* that is also provided by *OpenSS7*, and the *f_inode* pointer of the file pointer is adjusted to point directly to this `specfs inode`. This `specfs inode` is equivalent to the common `snode` used by *UNIX System V Release 4.2*.

Next, a *Stream header* is created from a `stdata(9)` data structure and a *Stream head* is created from a pair of `queue` structures. The content of the `stdata` data structure is initialized with predeter-

mined STREAMS values applicable to all character special *Streams*. The content of the `queue` data structures in the *Stream head* are initialized with values from the `streamtab` structure statically defined for *Stream heads* in the same manner as any STREAMS module or driver.

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

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

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

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

3.2.1.2 Subsequent Open of a Stream

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

3.2.2 Opening a STREAMS-based FIFO

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

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

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

[3] This is different that the situation in the *UNIX System V Release 4.2* system and other *UNIX* variants in the following respects: In *SVR 4.2* all FIFOs are STREAMS-based. In other *UNIX* implementations FIFOs are either *SVR 3.2*-style or, in some systems, optionally STREAMS-based. In *SVR 4.2* FIFOs are FIFO special files. In other *UNIX* implementations, FIFOs are character special files. Under *GNU/Linux*, system FIFOs are by default *SVR 3.2*-style FIFOs. To acheive the greatest possible degree of compatibility, *OpenSS7* provides the option of making all *GNU/Linux* system FIFOs STREAMS-based, and also provides a character special file implementation of STREAMS-based FIFOs.

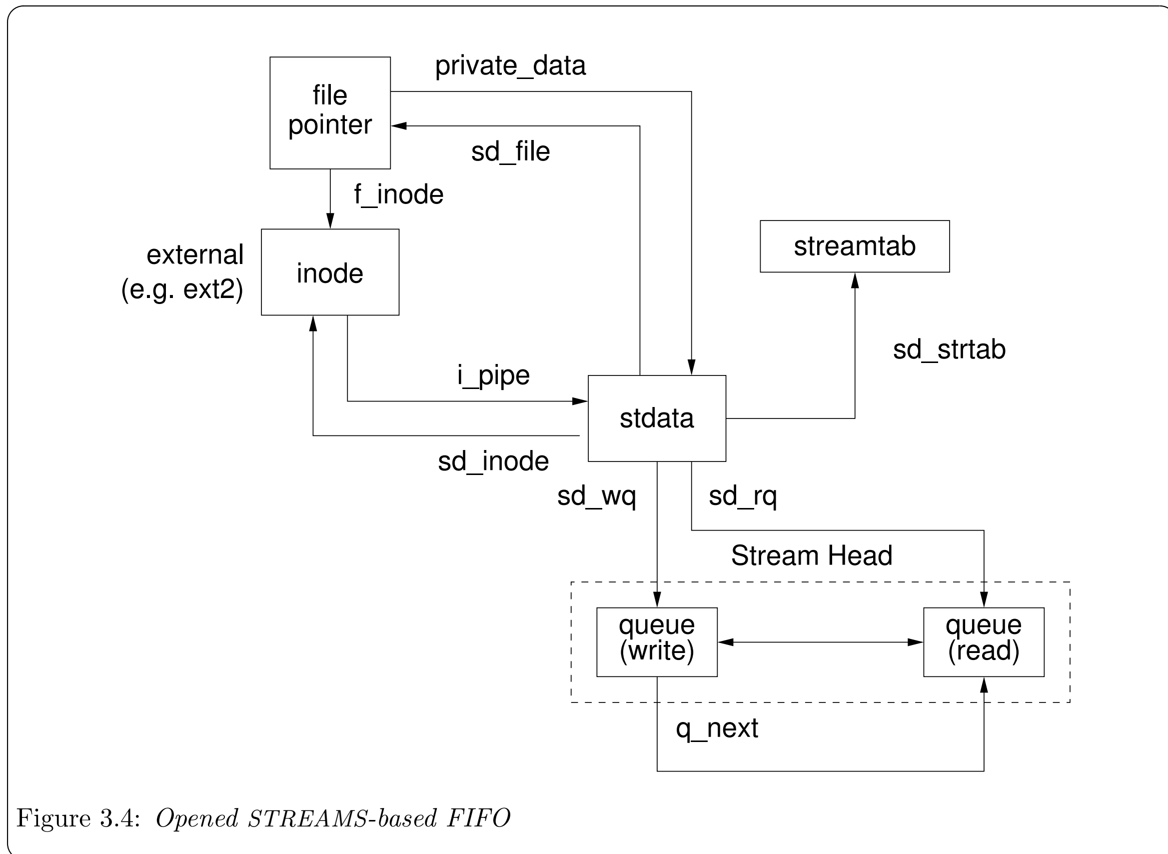

Figure 3.4: *Opened STREAMS-based FIFO*

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

1. When the *Stream header* (`stdata` structure) is created, it is attached to the external *GNU/Linux* file system `inode` instead of an `inode` within the *Shadow Special File System (specfs)*. This is also true of the `file` pointer: that is, the `file` pointer refers to the external file system `inode` instead of a `specfs inode`. The result is illustrated in Figure 3.4.

2. The *Stream header* (`stdata` structure) is initialized with limits and values appropriate for a STREAMS-based FIFO rather than a regular STREAMS driver. This is because the behaviour of a STREAMS-based FIFO *Stream head* must be somewhat different from a regular STREAMS driver to be compliant with *POSIX*.[4]

3. No driver queue pair is created or attached to the *Stream*. The *Stream head* write-side queue *q_next* pointer is set to the read-side queue as illustrated in Figure 3.4.

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

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

3.2.3 Creating a STREAMS-based Pipe

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

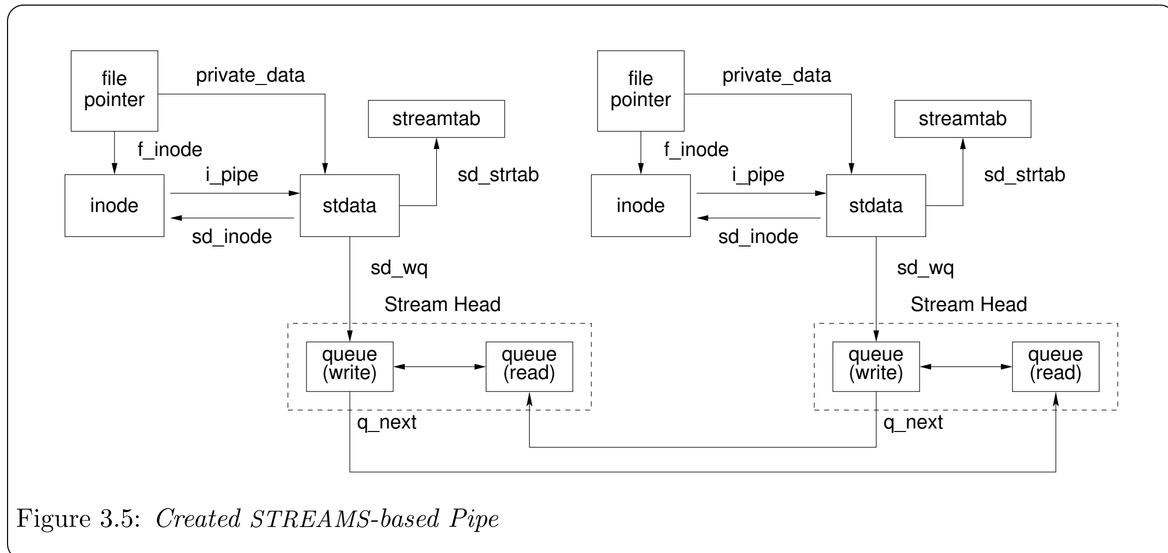

Figure 3.5: *Created STREAMS-based Pipe*

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

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

[5] Note that, by default, *GNU/Linux* system pipes obtained with the `pipe(2s)` system call are *SVR 3.2*-style unidirectional pipes. *OpenSS7* provides a `pipe(2s)` library function in the `libstreams` library that can be used to override the normal `pipe(2s)` system call for some applications programs. Also, *OpenSS7* provides the option of overriding all system pipes returned by the `pipe(2s)` system call to be bidirectional STREAMS-based pipes.

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

[7] Some *UNIX* implementations, and *UNIX System V Release 4*, provide a separate file system, the `pipefs`, upon which `vnodes` are created. In a simlar fashion, *GNU/Linux SVR 3.2*-style system pipes also allocates `inode`s from a `pipefs` file system.

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

- Instead of creating a driver queue pair for the *Stream*, the *q_next* pointer for the write-side `queue` of each *Stream head* is initialized to point to the read-side queue of the other *Stream head*. This is illustrated in Figure 3.5.

3.2.4 Adding and Removing Modules

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

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

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

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

3.2.4.1 Pushing Modules

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

Next, STREAMS positions the module's queue pair in the *Stream* immediately beneath the *Stream head* and above the driver and all existing modules on the *Stream*. Then the module's `open` procedure is called for the queue pair. (The `open` procedure is located in the *qi_qopen* member of the `qinit` structure associated with the read-side queue.)

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

To assist in identifying misbehaving applications programs that might push the same set of modules in an indefinite loop, swallowing an excessive amount of system resources, STREAMS imposes a limit on the number of modules that can be pushed on a given *Stream* to a practical number. The number is limited by the *NSTRPUSH* kernel parameter (see Appendix E [Configuration], page 139) which is set to either '`16`' or '`64`' on most systems.

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

3.2.4.2 Popping Modules

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

3.2.5 Closing the Stream

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

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

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

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

   When each module or driver queue pair is deallocated, any messages that remain on the queue are flushed prior to deallocation. Note that STREAMS frees only the messages contained on a message queue: any message or data structures used internally by the driver or module must be freed by the driver or module before it returns from its `close` procedure.
5. The queue pair associated with the *Stream head* is closed[10] and the queue pair and *Stream header* (`stdata` structure) are deallocated and the associated `inode`, `file` pointer, and file descriptors are released.

3.2.6 Stream Construction Example

This *Streams* construction example builds on the previous example (see Listing 1.1 in Section 1.3 [Basic Streams Operations], page 19), by adding the pushing of a module onto the open *Stream*.

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

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

3.2.6.1 Inserting Modules

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

The declarations for the user program are shown in Listing 3.1.

```
#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[BUFLE];
    int fd, count;
    struct strioct1 strioctl;
```

Listing 3.1: *Inserting Modules Example*

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

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

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

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

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

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

The difference in creating an instance of a STREAMS driver and module are illustrated in Listing 3.2 and Listing 3.3. An instance of a driver is created with the `open(2s)` system call, and each driver requires at least one device node in a file system for access. Naming of device nodes follow device naming conventions. On the other hand, an instance of a module is created with the `I_PUSH(7)` `ioct(2)` command. No file system device node is required. Naming of modules is separate from any file system considerations, and are chosen by the module writer. The only restrictions on a module name is that it be less than `FMNAMESZ` in length, and that it be unique.

When successful, the `I_PUSH(7)` `ioctl(2s)` call directs STREAMS to locate and insert the STREAMS module named `chconv` onto the *Stream*. If the `chconv` module has not been loaded into the *Linux* kernel, *OpenSS7* will attempt to demand load the kernel module named `streams-chconv`. Once the `chconv` STREAMS module is loaded in the kernel, STREAMS will create a queue pair for the instance of the module, insert it into the *Stream* beneath the *Stream head*, and call the module's `open` procedure. If the module's `open` procedure returns an error (typically only `[ENXIO]`), that error will be returned to the `ioctl(2s)` call. If the module's `open` procedure is successful, it (and the `ioctl(2s)` call), return '`0`'. The resulting *Stream* configuration is illustrated in Figure 3.6.

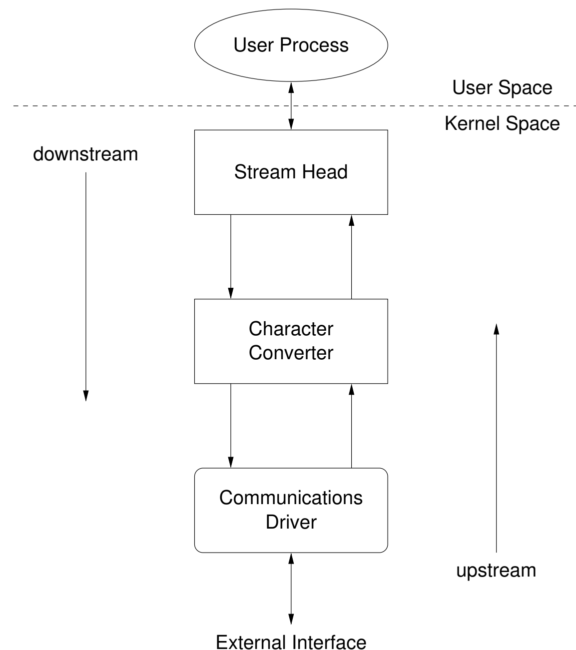

Figure 3.6: *Case Converter Module*

Modules are always pushed and popped from the position immediately beneath the *Stream head* in the manner of a push-down stack. This results in a *Last-In-First-Out (LIFO)* order of modules being

pushed and popped. For example, if another module were to be pushed on the *Stream* illustrated in Figure 3.6, it would be placed between the *Stream head* and the *Character Converter* module.

3.2.6.2 Module and Driver Control

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

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

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

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

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

Listing 3.4: *Module and Driver Control Example*

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

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

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

  It is this later method that illustrated in Listing 3.4.

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

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

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

In the Listing 3.4, two commands are issued to the character conversion module, `XCASE` and `DELETE`.[11]

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

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

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

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

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

3.2.6.3 Stream Dismantling with Modules

As shown in Listing 3.5, the remainder of this example follows the example in Listing 1.1 in Section 1.3 [Basic Streams Operations], page 19: data is read from the *Stream* and then echoed back to the *Stream*.

[11] These commands are fictitious.

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

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

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

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

3.2.6.4 Stream Construction Example Summary

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

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

4 Processing

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

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

4.1 Procedures

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

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

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

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

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

4.1.1 Put Procedure

The `put` procedure is invoked whenever a message is passed to a queue. A message can be passed to a queue using the `put(9s)`, `putnext(9)`, `putctl(9)`, `putctl1(9)`, `putctl2(9)`, `putnextctl(9)`, `putnextctl1(9)`, `putnextctl2(9)`, `qreply(9)` STREAMS utilities. The *Stream head*, modules and drivers use these utilities to deliver messages to a queue.[2] Invoking the `put` procedure of a queue with one of these utilities is the only accepted way of passing a message to a queue.[3]

A queue's `put` procedure is specified by the *qi_putp* member of the `qinit(9)` structure associated with the `queue(9)`. This is illustrated in Figure 4.1. In general, the read- and write-side queues of

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

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

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

a module or driver have different `qinit(9)` structures associated with them as there are differences in upstream and downstream message processing; however, it is possible for read- and write-side queues to share the same `qinit(9)` structure.

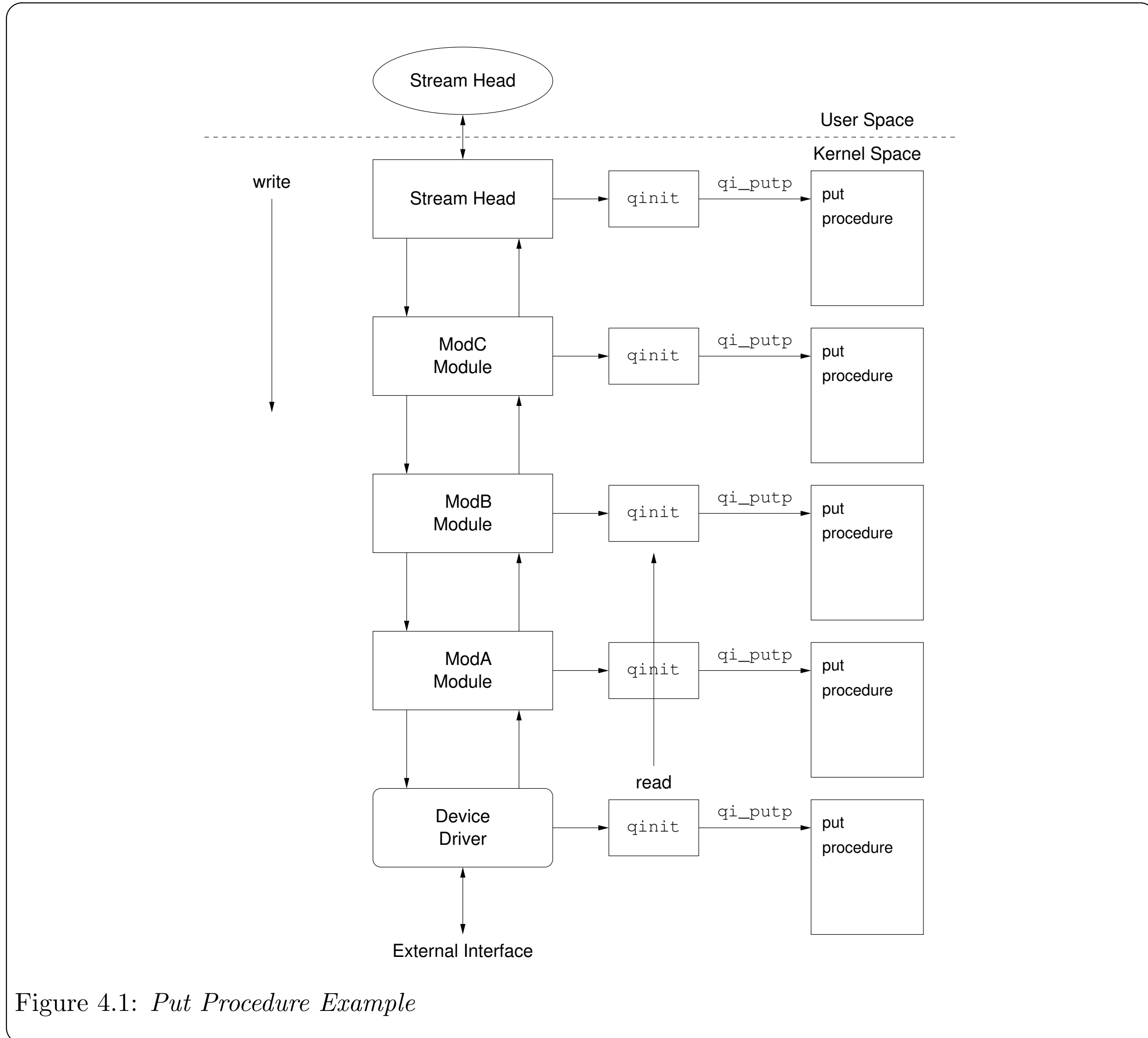

Figure 4.1: *Put Procedure Example*

The `put` procedure processes a message immediately or places it onto the message queue for later processing (generally by the `service` procedure). Because the `put` procedure is invoked before any queueing takes place, it provides a processing point at which the module or driver can take actions on time critical messages. `put` procedures are executed a higher priority than `service` procedures. `put` procedures in the upstream direction may even be executed with interrupts disabled.

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

When a number of modules are present in a *Stream*, as illustrated in Figure 4.1, each successive direct invocation of a `put` procedure is nested inside the others. For example, if the `put` procedure on the read-side of the driver is invoked by calling `put(9s)` from the driver's interrupt service routine,

and then each successive `put` procedure calls `putnext(9)`, by the time that the message reaches the *Stream head*, the driver, '`ModA`', '`ModB`', '`ModC`', and the *Stream head* `put` procedures will be nested within another.

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

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

4.1.2 Service Procedure

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

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

A queue's `service` procedure is never invoked directly by an adjacent module or driver. Adjacent modules or drivers invoke a queue's `put` procedure. The `put` procedure can then defer processing to the `service` procedure in a number of ways. The most direct way that a `put` procedure can invoke a `service` procedure for a message is to place that message on the message queue using `putq(9)`. Once the message is placed on the message queue in this manner, the `put` procedure can return, freeing the associated stack frame. Also, placing a message on the message queue with `putq(9)` will normally result in the queue's `service` procedure being *scheduled* for later execution by the STREAMS scheduler.

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

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

Processing of messages within a queue `service` procedure is performed by taking messages off of the message queue and processing them in order. Because messages a queued on the message queue with consideration to the priority class of the message, messages of higher priority are processed by the

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

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

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

4.1.3 Put and Service Procedure Summary

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

`put` and `service` procedures are addressed in more detail under Chapter 7 [Modules and Drivers], page 89. Design guidelines for `put` and `service` processing are given in ⟨undefined⟩ [⟨undefined⟩], page ⟨undefined⟩, ⟨undefined⟩ [⟨undefined⟩], page ⟨undefined⟩, and ⟨undefined⟩ [⟨undefined⟩], page ⟨undefined⟩.

4.2 Asynchronous Example

5 Messages

5.1 Messages Overview

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

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

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

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

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

5.1.1 Message Types

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

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

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

5.1.1.1 Ordinary Messages

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

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

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

The following message types are *not* defined by *SVR 4.2* and are *OpenSS7* specific, or are specific to another *SVR 4.2*-based implementation:

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

Ordinary messages are described in detail throughout this chapter and in Appendix B [Message Types], page 133.

5.1.1.2 High Priority Messages

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

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

M_IOCACK	-	U	ME	h	Positive ioctl(2s) acknowledgement
M_IOCNAK	-	U	ME	h	Negative ioctl(2s) acknowledgement
M_PCPROTO	D	U	HME	hme	Protocol control information
M_PCSIG	-	U	ME	h	Signal sent from a module/driver to a user
M_READ	D	-	H	me	Read notification, sent downstream
M_FLUSH	D	U	HME	hme	Flush module queue
M_STOP	D	-	ME	me	Suspend output
M_START	D	-	ME	me	Restart stopped device output
M_HANGUP	-	U	ME	h	Set a *Stream head* hangup condition, sent upstream
M_ERROR	-	U	ME	h	Report downstream error condition, sent upstream
M_COPYIN	-	U	ME	h	Copy in data for transparent[2] ioctls, sent upstream

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

[2] Transparent ioctls support applications developed prior to the introduction of STREAMS.

M_COPYOUT	-	U	ME	h	Copy out data for transparent[3] ioctls, sent upstream
M_IOCDATA	D	-	H	me	Data for transparent[4] ioctls, sent downstream
M_PCRSE	D	U	ME	hme	Reserved for internal use
M_STOPI	D	-	ME	me	Suspend input
M_STARTI	D	-	ME	me	Restart stopped device input

The following message types are *not* defined by *SVR 4.2* and are *OpenSS7* specific, or are specific to another *SVR 4.2*-based implementation:

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

High Priority messages are described in detail throughout this chapter and in Appendix B [Message Types], page 133.

5.1.2 Expedited Data

5.2 Message Structure

STREAMS messages consist of a chain of one or more message blocks. A message block is a triplet of a msgb(9) structure, a datab(9) structure, and a variable length data buffer. A message block (msgb(9) structure) is an instance of a reference to the data contained in the data buffer. Many message block structures can refer to a data block and data buffer. A data block (datab(9) structure) contains information not contained in the data buffer, but directly associated with the data buffer (e.g., the size of the data buffer). One and only one data block is normally associated with each data buffer. Data buffers can be internal to the message block, data block, data buffer triplet, automatically allocated using kmem_alloc(9), or allocated by the module or driver and associated with a data block (i.e., using esballoc(9)).

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

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

[3] Ibid.

[4] Ibid.

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

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

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

> Note that in *System V Release 4.0*, certain data structures fundamental to the kernel (for example, device numbers, user IDs) were enlarged to enable them to hold more information. This feature was referred to as Expanded Fundamental Types (EFT). Since some of this information was passed in STREAMS messages, there was a binary compatibility issue for pre-*System V Release 4* drivers and modules. `#ifdef`'s were added to the kernel to provide a transition period for these drivers and modules to be recompiled, and to allow it to be built to use the pre-*System V Release 4* short data types or the *System V Release 4* long data types. Support for short data types will be dropped in some future releases.[6]

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

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

The following flags are defined by *SVR 4.2*:

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

[5] Note that *OpenSS7* does not include the *b_pad2* member to reduce the size of the triplet and provide more room for a cache-aligned internal data buffer.
[6] System V Release 4 Programmer's Guide: STREAMS.

The following flags are *not* defined by *SVR 4.2* and are *OpenSS7* specific, or are specific to another *SVR 4.2*-based implementation:

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

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

typedef struct datab {
        union {
                struct datab *freep;
                struct free_rtn *frtnp;
        } db_f;
        unsigned char *db_base;
        unsigned char *db_lim;
        unsigned char db_ref;
        unsigned char db_type;
        unsigned char db_class;
        unsigned char db_pad;
        unsigned int db_size;
#if 0
        unsigned char db_cache[DB_CACHESIZE];
#endif
#if 0
        unsigned char *db_msgaddr;
        long db_filler;
#endif
        /* Linux Fast-STREAMS specific members */
        atomic_t db_users;
} dblk_t;

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

The following members are defined by *SVR 4.2*:

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

db_size size of the buffer;
db_cache SVR 3.1 internal buffer;[7]
db_msgaddr pointer to `msgb(9)` structure allocated with this data block in a 3-tuple;[8]
db_filler filler; and,[9]

The following members are *not* defined by *SVR 4.2* and are *OpenSS7* specific:

db_users same as *db_ref* but atomic.

5.2.1 Message Linkage

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

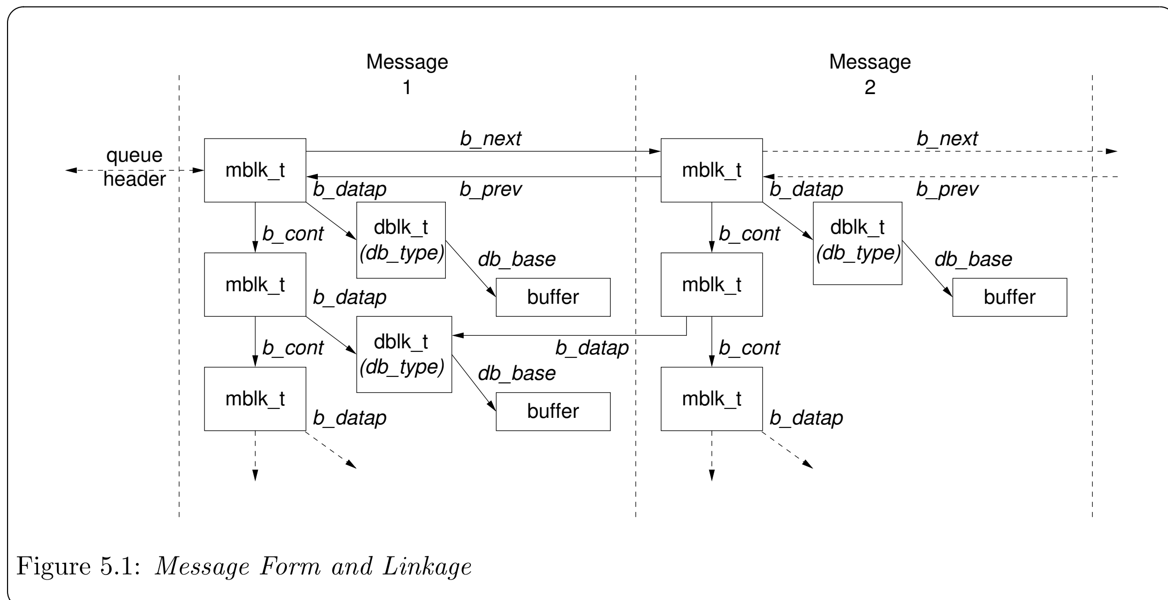

Figure 5.1: *Message Form and Linkage*

A message can occur stand-alone (that is, it is not present on any message queue as it is in a module or driver's `put` procedure) or can be queued on a message queue awaiting processing by the queue's

[7] This is an old SVR 3.1 member that was used to contain the internal data buffer. It is not longer at this location and this member is not present in *OpenSS7*.

[8] This member is used by some implementations to locate the initial `msgb(9)` structure allocated with this data block as a 3-tuple. *OpenSS7* calculates this address from the address of the data block itself and discards this member to reduce the overall size of the 3-tuple and to increase the cache-aligned size of the internal data buffer.

[9] *OpenSS7* discards this field to reduce the overall size of the structure and to increase the cache-aligned size of the internal data buffer.

`service` procedure. The *b_next* and *b_prev* pointers are not significant for a stand-alone message and are initialized to `NULL` by STREAMS when the message is not queued on a message queue.

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

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

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

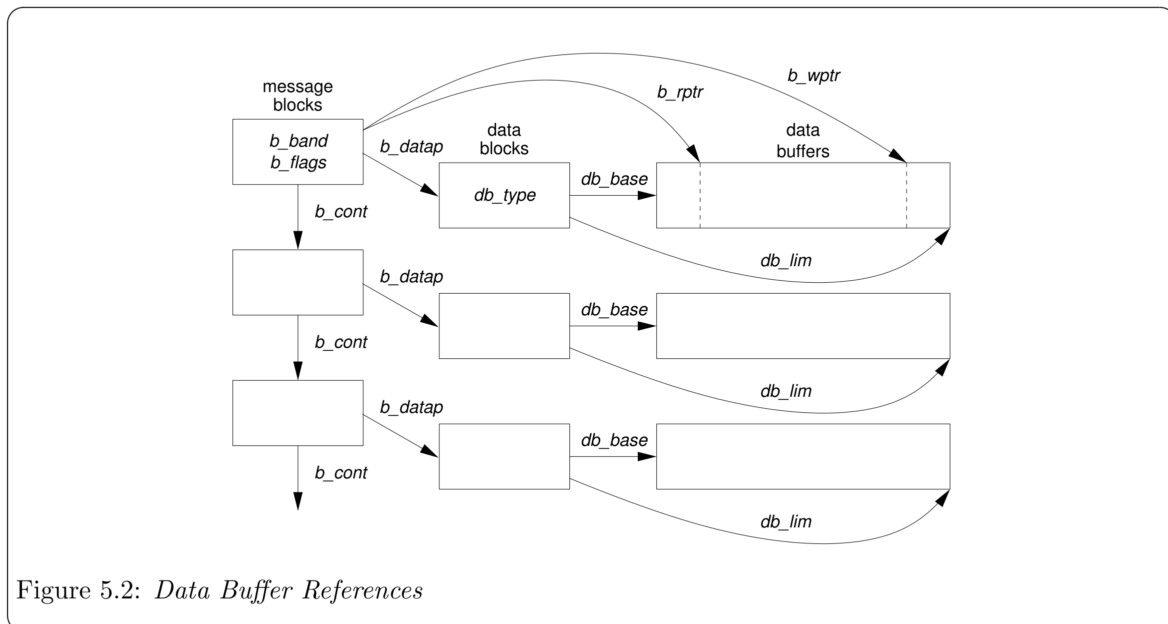

Figure 5.2: *Data Buffer References*

Being a reference to a data buffer, the message block has two pointer into the data buffer that define the range of data used by the reference. The *b_rptr* indicates the beginning of the range of data in the data buffer, and represents the position at which a module or driver would begin reading data; the *b_wptr*, the end of the range of data, where a module or driver would begin writing data. The data block, on the other hand, has two pointers representing the absolute limits of the data buffer. The *db_base* indicates the beginning of the data buffer; *db_lim*, the end. This relationship between pointers into the data buffer is illustrated in Figure 5.2.

STREAMS provides a library of utility functions used to manipulate message blocks, data blocks and data buffers. The members of a message block or data block should not be manipulated directly by the module or driver write: an appropriate STREAMS message utility should be used instead. See Appendix C [Utilities], page 135.

5.2.2 Sending and Receiving Messages

As shown in the message lists of Section 5.1 [Messages Overview], page 61, a large subset of the available message types can be generated and consumed by modules and drivers. Another subset, are dedicated to generation and consumption by the *Stream head*.

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

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

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

The STREAMS-specific `putmsg(2s)`, `putpmsg(2s)` system calls provide the user level process with the ability to directly generate `M_PROTO`, `M_PCPROTO` or `M_DATA` messages and have send downstream on the *Stream* from the *Stream head*. `getmsg(2s)`, `getpmsg(2s)` system calls provide the ability to collect `M_PROTO`, `M_PCPROTO` and `M_DATA` messages from the *Stream head*. These system calls are superior to the `write(2s)` and `read(2s)` system calls in that the provide a finer control over the composition of the generated message, and more information concerning the composition of a consumed message. Whereas, `write(2s)` and `read(2s)` pass only one buffer from the user, `putmsg(2s)`, `putpmsg(2s)`, `getmsg(2s)`, `getpmsg(2s)` provide two buffers: one for the control part of the message to transfer `M_PROTO` or `M_PCPROTO` message blocks with preservation of boundaries; another for the data part, to transfer `M_DATA` messages blocks – all in a single call. Also, data transfer with `write(2s)` and `read(2s)` are by nature byte-stream oriented, whereas, control and data transfer with `putmsg(2s)` and `getmsg(2s)` are by nature message oriented. `write(2s)` and `read(2s)` provide no mechanism for assigning priority to messages generated or indicating the priority of messages to be received: `putpmsg(2s)` and `getpmsg(2s)` provide the ability to specify criteria for the band (*b_band*) of the generated or consumed message.

---

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

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

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

5.2.2.1 putmsg(2s)

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

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

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

Where the arguments are interpreted as follows:

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

ctlptr is a pointer to a read-only `strbuf(5)` structure that is used to specify the control part of the message.

dataptr is a pointer to a read-only `strbuf(5)` structure that is used to specify the data part of the message.

flags specifies whether the control part of the message is to be of type `M_PROTO` or of type `M_PCPROTO`. It can have values '`0`' (specifying that an `M_PROTO` message be generated) or '`RS_HIPRI`' (specifying that an `M_PCPROTO` message be generated).

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

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

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

maxlen specifies the maximum length of the buffer and is ignored by `putmsg(2s)`;

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

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

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

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

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

5.2.2.2 getmsg(2s)

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

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

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

Where the arguments are interpreted as follows:

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

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

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

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

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

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

maxlen specifies the maximum length of the buffer into which the message part is to be written;

len ignored by getmsg(2s) on call, but set on return to indicate the length of the data that was actually written to the buffer by getmsg(2s); and,

buf specifies the location of the data buffer to contain the data retrieved for the control or data part of the message.

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

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

5.2.2.3 putpmsg(2s)

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

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

The arguments to `putpmsg(2s)` are interpreted the same as those for `putmsg(2s)` as described in Section 5.2.2.1 [putmsg(2s)], page 69 with the exception of the *band* and *flags* arguments.

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

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

Under *OpenSS7*, `putmsg(2s)` is implemented as a library call to `putpmsg(2s)`. This is possible because the call:

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

is equivalent to:

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

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

5.2.2.4 getpmsg(2s)

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

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

The arguments to `getpmsg(2s)` are interpreted the same as those for `getmsg(2s)` as described in Section 5.2.2.2 [getmsg(2s)], page 70, with the exception of the *bandp* and *flags* arguments.

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

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

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

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

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

Under *OpenSS7*, `getmsg(2s)` is implemented as a library call to `getpmsg(2s)`. This is possible because the calls:

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

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

are equivalent to:

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

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

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

5.2.3 Control of Stream Head Processing

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

Modules and drivers can control *Stream head* processing using the `M_SETOPTS` message. At any time, a module or driver can issue an `M_SETOPTS` message upstream. The `M_SETOPTS` contains a `stroptions(9)` structure (see Appendix A [Data Structures], page 131) specifying which *Stream head* characteristics to alter in the read-side queue of the *Stream head* (including *q_hiwat*, *qi_lowat*, *q_minpsz* and *q_maxpsz*), however, of interest to the current discussion are the read and write options associated with the *Stream head*.

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

5.2.3.1 Read Options

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

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

5.2.3.2 Read Mode

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

`RNORM`
: byte-stream mode. This is the default read mode. This is the normal byte-stream mode where message boundaries are ignored. `read(2s)` and `readv(2s)` return data until the read count has been satisfied or a zero length message is received.

`RMSGD`
: message non-discard mode. The `read(2s)` and `readv(2s)` system calls will return when either the count is satisfied, a zero length message is received, or a message boundary is encountered. If there is any data left in a message after the read count has been satisfied, the message is placed back on the *Stream head* read queue. The data will be read on a subsequent `read(2s)` or `readv(2s)` call.

`RMSGN`
: message discard mode. Similar to `RMSGN` mode, above, but data that remains in a message after the read count has been satisfied is discarded.

RFILL — message fill mode. Similar to RNORM but requests that the *Stream head* fill a buffer completely before returning to the application. This is used in conjunction with a cooperating module and M_READ messages.[13]

5.2.3.3 Read Protocol

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

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

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

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

RPROTCOMPRESS — compress like data.[18]

Note that, although all modes terminate the read on a zero-length message, *POSIX* requires that zero only be returned from read(2s) when the requested length is zero or an end of file (M_HANGUP) has occurred. Therefore, *OpenSS7* only returns on a zero-length message if some data has been read already.

5.2.3.4 Write Options

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

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

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

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

---

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

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

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

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

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

[18] The RPROCOMPRESS option is not defined by *SVR 4.2*, but is defined by some implementations based on *SVR 4.2*.

`SNDHOLD` Requests that the *Stream head* hold messages temporarily in an attempt to coalesce smaller messages into larger ones for efficiency. This feature is largely deprecated, but is supported by *OpenSS7*. When not set (as is the default), messages are sent immediately. This option is *not* set by default for any *Stream*.

5.2.3.5 Write Offset

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

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

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

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

5.3 Queues and Priority

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

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

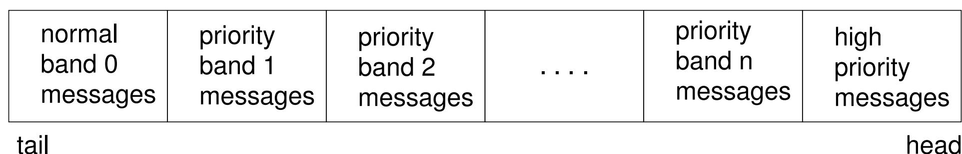

Figure 5.3: *Message Ordering on a Queue*

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

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

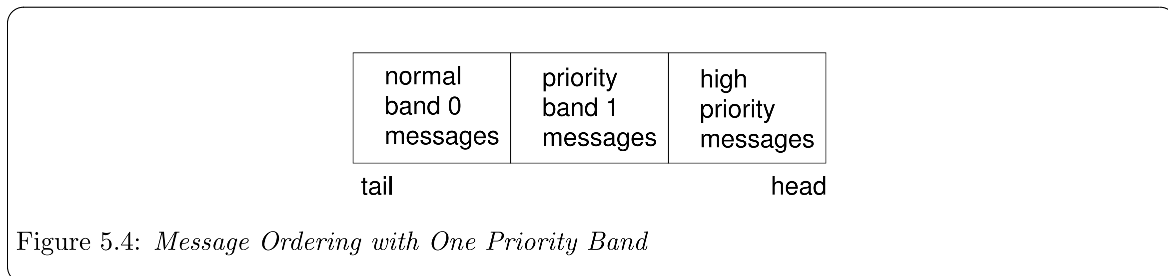

Figure 5.4: *Message Ordering with One Priority Band*

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

5.3.1 Queue Priority Utilities

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

flushq(9)
flushband(9)
: These utilities provide the ability to flush specific messages from a message queue. They are discussed under Section 7.3 [Flush Handling], page 89, and under Appendix C [Utilities], page 135. These utilities are also described in the corresponding manual page.

canput(9)
bcanput(9)
canputnext(9)
bcanputnext(9)
: These utilities provide the ability to test the current or next queue for a flow control condition for normal (band zero) messages or priority messages within a message

band. They are discussed under Section 5.3.5.1 [Flow Control], page 84, and under Appendix C [Utilities], page 135. These utilities are also described in the corresponding manual page.

`strqset(9)`
`strqget(9)`
: These utilities provide the ability to examine and modify flow control parameters associated with a queue (`queue(9)`) or queue band (`qband(9)`). They are discussed below, and under Appendix C [Utilities], page 135. These utilities are also described in the corresponding manual page.

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

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

5.3.1.1 strqget(9)

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

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

Where the arguments are interpreted as follows:

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

what
: Specifies which member to retrieve. Specific values for various members are described below.

band
: When zero, specifies that the member is to be retrieved from the `queue(9)` structure specified by *q*; when non-zero, the band number of the `qband(9)` structure from which to retrieve the member.

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

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

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

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

`QHIWAT` Set or return the high water mark (*q_hiwat* or *qb_hiwat*).

`QLOWAT` Set or return the low water mark (*q_lowat* or *qb_lowat*).

`QMAXPSZ` Set or return the maximum packet size (*q_maxpsz* or *qb_maxpsz*).

`QMINPSZ` Set or return the minimum packet size (*q_minpsz* or *qb_minpsz*).

`QCOUNT` Return the count of bytes queued (*q_count* or *qb_count*). This field is only valid for `strqget(9)`.

`QFIRST` Return a pointer to the first message queued (*q_first* or *qb_first*). This field is only valid for `strqget(9)`.

`QLAST` Return a pointer to the last message queued (*q_last* or *qb_last*). This field is only valid for `strqget(9)`.

`QFLAG` Return the flags word (*q_flag* or *qb_flag*). This field is only valid for `strqget(9)`.

Additional information is given under Appendix C [Utilities], page 135, and provided in the `strqget(9)` manual page.

5.3.1.2 strqset(9)

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

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

Where the arguments are interpreted as follows:

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

what Specifies which member to write. Specific values for various members are described above under Section 5.3.1.1 [strqget(9)], page 76.

band When zero, specifies that the member is to be written to the `queue(9)` structure specified by *q*; when non-zero, the band number of the `qband(9)` structure to which to write the member.

val Specifies the `long` value to write to the member. All values are converted to a `long` to be passed in this argument.

Additional information is given under Appendix C [Utilities], page 135, and provided in the `strqset(9)` manual page.

5.3.2 Queue Priority Commands

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

I_FLUSHBAND(7)
: Flushes the *Stream* for a specified band. This ioctl(2s) command is equivalent to the flushq(9) and flushband(9) utilities available to modules and drivers. It is discussed under Section 7.3 [Flush Handling], page 89.

I_CKBAND(7)
: Checks whether a message is available to be read from a specified queue band. It is discussed below.

I_GETBAND(7)
: Gets the priority band associated with the next message on the *Stream head* read queue. It is discussed below.

I_CANPUT(7)
: Checks whether messages can be written to a specified queue band. This ioctl(2s) command is equivalent to the canput(9) and bcanput(9) utilities available to modules and drivers. It is discussed under Section 5.3.5.1 [Flow Control], page 84.

I_ATMARK(7)
: This ioctl(2s) command supports *Transmission Control Protocol (TCP)* urgent data in a byte-stream. It indicates when a marked message has arrived at the *Stream head*. It is discussed below.

I_GETSIG(7)
I_SETSIG(7)
: Sets the mask of events for which the *Stream* head will send a calling process a {SIGPOLL} or {SIGURG} signal. Events include S_RDBAND, S_WRBAND and S_BANDURG. This ioctl(2s) command is discussed under Section 6.1 [Input and Output Polling], page 87.

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

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

5.3.2.1 I_FLUSHBAND

Flushes the *Stream* for a specified band. This ioctl(2s) command is equivalent to the flushq(9) and flushband(9) utilities available to modules and drivers. It is discussed under Section 7.3 [Flush Handling], page 89.

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

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

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

where,

bi_pri the priority band to flush;

bi_flag how to flush: one of `FLUSHR`, `FLUSHW` or `FLUSHRW`.

5.3.2.2 `I_CKBAND`

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

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

cmd is '`I_CKBAND`'.

arg contains the band number for which to check for an available message.

5.3.2.3 `I_GETBAND`

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

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

cmd is '`I_GETBAND`'.

arg is a pointer to an `int` into which to receive the band number.

5.3.2.4 `I_CANPUT`

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

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

where,

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

cmd is '`I_CANPUT`'.

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

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

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

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

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

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

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

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

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

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

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

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

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

OpenSS7 implements the special flag, `ANYBAND`, that can be used for an *arg* value instead of the band number to check whether any existing band is writable. This is similar to the `POLLWRBAND` flag to `poll(2s)`. `ANYBAND` uses the otherwise invalid band number '`-1`'. Portable STREAMS applications programs will not use the `ANYBAND` flag and will not rely upon `I_CANPUT(7)` to generate an error if passed '`-1`' as an invalid argument.

5.3.2.5 `I_ATMARK`

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

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

where,

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

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

`ANYMARK`
: Determine if the message at the head of the *Stream head* read queue is marked by a downstream module or driver.

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

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

STREAMS message blocks that have the `MSGMARK` flag set in the *b_flag* member of the `msgb(9)` structure are marked messages. *Solaris* also provides the `MSGMARKNET` and `MSGNOTMARKNET` flags. The use of these flags is not very clear, but *OpenSS7* could use them in the `read(2s)` logic to determine whether the next message is marked without removing the message from the queue.

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

The *b_flag* member of the `msgb(9)` structure can have the flag, `MSGMARK`, set that allows a module or driver to mark a message sent to the *Stream head*. This is used to support `tcp(4)`'s ability to indicate the last bye of out-of-band data. Once marked, a message sent to the *Stream head* causes the *Stream head* to remember the message. A user may check to see if the message on the front of the *Stream head* read queue is marked, and whether it is the last marked message on the queue, with the `I_ATMARK(7)` `ioctl(2s)` command. If a user is reading data from the *Stream head* and there are multiple messages on the *Stream head* read queue, and one of those messages is marked, `read(2s)` terminates when it reaches the marked message and returns the data only up to that marked message. The rest of the data may be obtained with successive reads. `ANYMARK` indicates that the user merely wants to check if the message at the head of the *Stream head* read queue is marked. `LASTMARK` indicates that the user wants to see if the message is the only one marked on the queue.

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

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

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

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

5.3.2.6 `I_GETSIG`

Sets the mask of events for which the *Stream* head will send a calling process a {`SIGPOLL`} or {`SIGURG`} signal. Events include `S_RDBAND`, `S_WRBAND` and `S_BANDURG`. This `ioctl(2s)` command is discussed under Section 6.1 [Input and Output Polling], page 87.

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

Event flags can include the following band related events:

`S_RDBAND` a message of non-zero priority band has been placed to the *Stream head* read queue.

`S_WRBAND` a priority band that was previously flow controlled has become available for writing (i.e., is no longer flow controlled).

`S_BANDURG` a modifier to `S_RDBAND` to generate {`SIGURG`} instead of {`SIGPOLL`} in response to the event.

5.3.2.7 `I_SETSIG`

Sets the mask of events for which the *Stream* head will send a calling process a {`SIGPOLL`} or {`SIGURG`} signal. Events include `S_RDBAND`, `S_WRBAND` and `S_BANDURG`. This `ioctl(2s)` command is discussed under Section 6.1 [Input and Output Polling], page 87.

fildes the *Stream* for which the command is issued;
cmd is ‘`I_SETSIG`’.
arg is an integer value that contains the event flags.

Event flags can include the following band related events:

`S_RDBAND` a message of non-zero priority band has been placed to the *Stream head* read queue.

S_WRBAND — a priority band that was previously flow controlled has become available for writing (i.e., is no longer flow controlled).

S_BANDURG — a modifier to S_RDBAND to generate {SIGURG} instead of {SIGPOLL} in response to the event.

5.3.3 The queue Structure

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

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

The following members are defined in *SVR 4.2*:

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

The following members are not defined in *SVR 4.2* and are *OpenSS7* specific:

q_msgs — number of messages on the queue;

q_lock	queue structure lock; and,
q_ftmsg	message filter ala AIX.

5.3.3.1 Using queue Information

5.3.3.2 queue Flags

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

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

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

The following are not defined by *SVR 4.2*, but are used by *OpenSS7* and other *SVR 4.2*-based implementations:

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

5.3.4 The qband Structure

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

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

#define qb_msgs qb_pad1
```

Where the members are interpreted as follows:

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

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

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

5.3.4.1 Using qband Information

5.3.5 Message Processing

5.3.5.1 Flow Control

5.3.6 Scheduling

5.3.6.1 Flow Control Variables

5.3.6.2 Flow Control Procedures

5.3.6.3 The STREAMS Scheduler

5.4 Service Interfaces

5.4.1 Service Interface Benefits

5.4.2 Service Interface Library Example

5.4.2.1 Accessing the Service Provider

5.4.2.2 Closing the Service Provider

5.4.2.3 Sending Data to the Service Provider

5.4.2.4 Receiving Data

5.4.2.5 Module Service Interface Example

5.5 Message Allocation

5.5.1 Recovering From No Buffers

5.6 Extended Buffers

6 Polling

6.1 Input and Output Polling

6.2 Controlling Terminal

7 Modules and Drivers

7.1 Environment

7.2 Input-Output Control

7.3 Flush Handling

7.4 Driver-Kernel Interface

7.5 Design Guidelines

8 Modules

8.1 Module

8.2 Module Flow Control

8.3 Module Design Guidelines

9 Drivers

9.1 External Device Numbers

9.2 Internal Device Numbers

9.3 spec File System

9.4 Clone Device

9.5 Named STREAMS Device

9.6 Driver

9.7 Cloning

9.8 Loop-Around Driver

9.9 Driver Design Guidelines

10 Multiplexing

10.1 Multiplexors

10.2 Connecting and Disconnecting Lower Stream

10.3 Multiplexor Construction Example

10.4 Multiplexing Driver

10.5 Persistent Links

10.6 Multiplexing Driver Design Guidelines

11 Pipes and FIFOs

11.1 Pipes and FIFOs

11.2 Flushing Pipes and FIFOs

11.3 Named Streams

11.4 Unique Connections

12 Terminal Subsystem

12.1 Terminal Subsystem

12.2 Pseudo-Terminal Subsystem

13 Synchronization

This chapter describes how to multi-thread a STREAMS driver or module. It covers the necessary conversion topics so that new and existing STREAMS modules and drivers will run in a symmetrical multi-processor kernel. This chapter covers primarily STREAMS specific multiprocessor issues and techniques.

Linux is a fully *SMP* capable operating system able to make effective use of the available parallelism of the symmetric shared-memory multiprocessor computer. All kernel subsystems are multiprocessor safe: scheduler, virtual memory, file systems, block, character, STREAMS input and output, networking protocols and device drivers.

STREAMS in an *MP* environment introduces some new concepts and terminology as follows:

Thread sequence of instructions executed within the context of a process

Lock mechanism for restricting access to data structures

Single Threaded
restricting access to a single thread

Multi-Threaded
allowing two or more threads access

Multiprocessing
two or more CPUs concurrently executing the operating system

The **Linux** 2.6 and 3.x 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 *Syncrhonization 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 *OpenSS7* subsystem:

1. *Synchronous.* These entry points (callouts) and callbacks 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(9s)` –
   `srv(9s)` –
   `qopen(9)` –
   `qclose(9)` –
   `qbufcall(9)` –
   `qtimeout(9)` –
   `mi_bufcall(9)` –
   `putq(9)` –
   `putbq(9)` –
   `putnext(9)` –
   `qreply(9)` –

2. *Asynchronous.* These callbacks are *not* referenced against a STREAMS queue structure. That is, they were invoked without a specific STREAMS queue structure as an argument (known to STREAMS). These procedures are as follows:

   `bufcall(9)` –

esbbufcall(9)	–
timeout(9)	–
esballoc(9)	(free routine)

13.1 MT 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`
: *Default level synchronization.* Specifies that the module uses the default synchronization scheme. This is the same as specifying `SQLVL_MODULE`.

`SQLVL_GLOBAL`
: *Global (STREAMS scheduler) level synchronization.* Specifies that all of STREAMS can be access by only one thread at the same time. The module is run with global synchronization. This means that only one STREAMS executive thread will be permitted to enter any module. This makes the entire STREAMS executive single threaded and is useful primarily for debugging. This is the same as "Uniprocessor Emulation" on some systems, and reduces the STREAMS executive to running on a single processor at a time. This option should normally be used only for debugging.

`SQLVL_ELSEWHERE`
: *Module group level synchronization.* 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`
: *Module level synchronization.* 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 STREAMS guidelines. This level is roughly equivalent to *Solaris* `D_MTPERMOD` perimeters.

`SQLVL_QUEUEPAIR`
: *Queue pair level synchronization.* 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 *Solaris* `D_MTAPAIR` perimeters.

`SQLVL_QUEUE`
: *Queue level synchronization.* 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 *Solaris* `D_MTPERQ` perimeters.

`SQLVL_NOP`
: *No synchronization.* Specifies that each queue can be accessed by more than one thread at a the same time. The protection of internal data and of `put(9s)` and `srv(9s)` procedures against `timeout(9)` or `bufcall(9)` 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 *Solaris* `D_MP` flag.

13.2 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(9s)`
: *Queue put procedure.* 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(9s)`
: If the module has any synchronization level other than `SQLVL_NOP`, *Queue service procedure.* 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)`
: *Queue open procedure.* The queue open procedure is synchronous and exclusive before the call to `qprocson(9)`, 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(9)` procedure is exclusive.

`qclose(9)`
: *Queue close procedure.* The queue close procedure is synchronous and exclusive after the call to `qprocsoff(9)`, 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 `qclose(9)` procedure is exclusive.

`qprocson(9)`
: *Queue procedures on.*

`qprocsoff(9)`
: *Queue procedures off.*

`freezestr(9)`
: *Freeze stream.*

`unfreezestr(9)`
: *Thaw stream.*

`qwriter(9)`
: *Queue writer.*

13.3 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:

`qbufcall(9)`	– queue referenced buffer call
`qtimeout(9)`	– queue referenced timeout
`qunbufcall(9)`	– queue referenced buffer call cancel
`quntimeout(9)`	– queue referenced timeout cancel
`mi_bufcall(9)`	– queue reference buffer call

13.4 Synchronous Callouts

`putnext(9)`	–
`qreply(9)`	–

13.5 Asynchronous Entry Points

13.6 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.

13.7 Asynchronous Callouts

13.8 STREAMS Framework Integrity

The STREAMS framework guarantees the integrity of the STREAMS scheduler and related data structures, such as the `queue(9)`, `msgb(9)`, and `datab(9)` structures, assuming that the module properly accesses global operating system data structures, utilities and facilities.

The *q_next* and *q_ptr* members of the `queue(9)` structure will not be modified by the system while a thread is actively executing within a synchronous entry point. The *q_next* member of the `queue(9)` structure could change while a thread is executing within an asynchronous entry point.

A STREAMS module or driver must not call another module's `put` or `service` procedure directly. The STREAMS utilities `putnext(9)`, `put(9s)` and others described in Appendix C [Utilities], page 135 must be used to pass messages to another queue. Calling another STREAMS module or driver directly circumvents the *MP-STREAMS* framework.[1]

To make a STREAMS module or driver *MP-SAFE* requires that the integrity of private module data structures be protected by the module itself. The integrity of private module data structures can be maintained either by using the *MP-STREAMS* framework to control concurrency and synchronize access to private data structures, or by the use of private locks within the module, or a combination of the two.

[1] The practise of calling a neighbouring module's put or service procedure directly using the *qi_putp* or *qi_srvp* members of the `qinit(9)` structure is long deprecated and has not been seen in drivers since SVR 3.

13.9 MP Message Ordering

STREAMS guarantees the ordering of messages along a *Stream* if all the modules in the *Stream* preserve message ordering internally. This ordering guarantee only applies to message that are sent along the same *Stream* and produced by the same source.

STREAMS does not guarantee that a message has been seen by the next `put` procedure by the time that `putnext(9)` or `qreply(9)` return. Under some circumstances, invocation of the next module's `put` procedure might be deferred until after an exclusive thread leaves a synchronization boundary.

Regardless of STREAMS integrity protection, or the presence of synchronization barriers, at most one thread will be executing a given module's `service` procedure.

13.10 MP-UNSAFE Modules

STREAMS supports modules that are not MP-SAFE and that are expecting to run in a uniprocessor environment.

By default, all STREAMS modules and drivers are considered *MP-UNSAFE* unless configured into the system as *MP-SAFE*.

Unsafe drivers run with only the minimum of modification. Unsafe drivers are synchronized, by default, at the level `SQLVL_MODULE`, which implies that, at any time, only one processor in the entire system is executing the module's STREAMS code. *MP-UNSAFE* modules might not gain any performance advantage by being run in a multiprocessor environment.

MP-UNSAFE modules that access data structures private to other STREAMS modules must be synchronized at a broader level of synchronization. All such cooperating modules must be run with synchronization at the level `SQLVL_ELSEWHERE`, with a synchronization queue that is shared across all the pertinent modules.

MP-UNSAFE modules that do not share data between Stream instances but do shared Stream private data between the read and write put and service procedures can be synchronized at level `SQLVL_QUEUEPAIR` and will gain some advantage in the multiprocessor environment.

MP-UNSAFE modules that do not share data between Stream instances and do not share data between read and write side put and service procedures, but do share data between put and service procedure on the same side, can be synchronized at level `SQLVL_QUEUE` and will gain some advantage in the multiprocessor environment.

MP-UNSAFE modules that shared data between Stream instances, but only in the open and close routines, can still assign `SQLVL_QUEUEPAIR` or `SQLVL_QUEUE`, provided that an outer barrier is also established using the *Solaris*®-style outer perimeter (with the `D_MTOCEXCL` flag).

13.10.1 MP-UNSAFE Open and Close Routines

MP-UNSAFE modules are still responsible for cancelling all outstanding callbacks in their *qi_qclose* procedure.

MP-UNSAFE modules that are synchronized at `SQLVL_QUEUEPAIR` or `SQLVL_QUEUE`, that do not have an exclusive outer perimited established with `D_MTOCEXCL`, must call `qprocsoff(9)` in the *qi_qclose* routine, in addition to cancelling all oustanding callbacks, before deallocating Stream private structures or altering *q_qptr* pointers.

13.10.2 MP-UNSAFE Put and Service Procedures

13.10.3 MP-UNSAFE Interrupt Service Routines

MP-UNSAFE modules synchronized at synchronization level `SQLVL_MODULE`, `SQLVL_ELSEWHERE`, or `SQLVL_GLOBAL` are singly threaded within the STREAMS framework. However, interrupt service

routines exist outside the STREAMS framework. Interrupt service routines that invoke STREAMS utilities will have execution of those utilities deferred until after all threads have left the synchronization barrier.

13.10.4 MP-UNSAFE Shared Data Structures

Modules that share data structure(s), and that are to be protected by STREAMS synchronization, must be configured at the same level of synchronization.

13.10.5 MP-UNSAFE Sleeping

An *MP-UNSAFE* module that must wait in its `open` or `close` procedure for a message from another STREAMS module must wait outside of all synchronization barriers; otherwise the responding thread might never be allowed to enter the synchronization barrier to invoke the module's `put` or `service` procedure. Sleeping outside the synchronization barriers is accomplished by using `qwait(9)` or `qwait_sig(9)`.

Modules using STREAMS synchronization barriers, either explicitly by configuration, or by default, must use `qwait(9)` and `qwait_sig(9)` instead of `CV_WAIT(9)` or `CV_WAIT_SIG(9)` from within *qi_qopen* and *qi_qclose* procedures.[2]

13.11 MP-SAFE Modules

13.11.1 MP Put and Service Procedures

The STREAMS utilities `qprocson(9)` and `qprocsoff(9)` enable and disable the `put` and `service` procedures of a queue pair. Prior to a call to `qprocson(9)` and after a call to `qprocsoff(9)`, the module's `put` and `service` procedures are disabled. Messages flow around the module as if it were not present in the *Stream*.

`qprocson(9)` must be called by the first `open(2s)` of a module, but only after allocation and initialization of any module resources or private data structures upon which the `put` and `service` procedures depend. `qprocsoff(9)` must be called by the `close(2s)` routine of a module before deallocating any resources on which the `put` and `service` procedures depend.

For example, it is typical for a module's *qi_qopen* procedure to allocate a private data structure and associate it with the read- and write-queue *q_ptr* pointer for use by both the `put` and `service` procedure. It is typical for a module's *qi_qclose* procedure to free the private data structure. In this case, `qprocson(9)` should not be called until *after* the private data structure has been allocated, initialized and attached to the *q_ptr* pointers. `qprocsoff(9)` should be called *before* deallocating the private data structure and invalidating the *q_ptr* pointers.

13.11.2 MP Timeout and Buffer Callbacks

The `timeout(9)`, `bufcall(9)` and `esbbcall(9)` callbacks are asynchronous when invoked from outside the STREAMS framework. The means that the `timeout(9)`, `bufcall(9)`, or `esbbcall(9)` callback functions might execute concurrent with module procedures.

In contrast, under *OpenSS7*, when `timeout(9)`, `bufcall(9)`, and `esbbcall(9)` are invoked from within the STREAMS framework,[3] they are equivalent to a call to `qtimeout(9)`, `qbufcall(9)` with

[2] Modules are not permitted to sleep outside of their queue open and close procedures. Attempting to sleep in a `put` or `service` procedure will panic most kernels.

[3] That is, they are invoked from a module's `put` or `service` procedure, or from within another synchronous callback, but not within a module's `open` or `close` procedures.

the current synchronization queue used as the *q* argument. This is possible because STREAMS always knows what queue's synchronous procedures or callbacks it is running.

To provide for synchronous callbacks that can be invoked from outside the STREAMS framework, the `qtimeout(9)`, `quntimeout(9)` `qbufcall(9)`, and `qunbufcall(9)` STREAMS utilities are provided. When using these utilities, the callback function is executed inside any synchronization barrier associated with the queue that is passed to the function.

There are some restrictions on which queue pointer the `qtimeout(9)` and `qbufcall(9)` can be passed when called from a module's `open` or `close` procedure, or when called from outside STREAMS (at soft or hard interrupt). The caller is responsible for the validity of the queue pointer. That is, the queue must be allocated and have procedures enabled across the call. The queue pointer argument of a module's `open`, `close`, `put`, or `service` procedure can always be passed as an argument to these functions without any special consideration. They should not be passed a *q->q_next* pointer, unless the *Stream* is first frozen by the caller with `freezestr(9)`. They may be passed a driver's read-side queue pointer, or a lower multiplexed *Stream*'s write-side queue pointer, provided that the caller can ensure that the driver is not closed and the multiplexed *Stream* is not unlinked across the call. Reference to interior queue pairs must not be performed unless the *Stream* has first been frozen by the caller with `freezestre(9)`.

13.11.3 MP Open and Close Procedures

STREAMS modules are permitted to sleep in their *qi_qopen* and *qi_qclose* procedures. However, MP-UNSAFE modules that use synchronization of these procedures against `put` and `service` procedures must leave the synchronization barrier before sleeping. This is accomplished by using the `qwait(9)` and `qwait_sig(9)` STREAMS utilities. These utilities are similar to `CV_WAIT(9)` and `CV_WAIT_SIG(9)`, however, they release the synchronization barrier before sleeping. These MP-UNSAFE utilities may also be used by MP-SAFE modules; however, MP-SAFE modules may also use `CV_WAIT(9)` or `CV_WAIT_SIG(9)`.

Because callback functions can be asynchronous with respect to the STREAMS framework, they might execute concurrent with a module's `close` procedure. It is the responsibility of the module to cancel all outstanding callbacks before deallocating or invalidating references to data structures upon which those callbacks depend, and before returning from the `close` procedure.

A callback function scheduled with `timeout(9)` or `bufcall(9)` are guaranteed to have been cancelled by the time that the corresponding `untimeout(9)` or `unbufcall(9)` utilities return. The same is true for `qtimeout(9)`, `qbufcall(9)`, `quntimeout(9)` and `qunbufcall(9)`.

The Mentat Portable Streams (MPS®) framework provided by the *STREAMS Compatibility Modules* package for *Linux Fast-STREAMS* also provides an `mi_bufcall(9)` function and `mi_timer(9)` function that can be used to manage buffer callbacks and timeouts as well as converting these asyncrhonous events into STREAMS synchronous events.

13.11.4 MP Module Unloading

STREAMS tracks kernel module references and prohibits a kernel module from unloading while there is a reference to a statically allocated data structure contained within the kernel module. If a STREAMS module does not cancel all callbacks in the module `close` procedure, the associated kernel module must not be permitted to be unloaded. STREAMS handles all references with the exception of references to the free routine provided to `esballoc(9)`.

STREAMS loadable kernel modules that pass free routines to `esballoc(9)` are responsible for incrementing their own module counts upon the call to `esballoc(9)` and decrementing them when the *free_rtn* function exits.[4]

13.11.5 MP Locking

Basic spin locks or reader/writer locks can be used by *MP-SAFE* modules to protect module private data structures. When using locks, however, the following guidelines should be followed:

- Avoid holding module private locks across calls to `putnext(9)`, `qreply(9)`, or other STREAMS utilities that invoke a `put` procedure, unless re-entrancy is provided. Otherwise, the calling thread might reenter the same queue procedure and attempt to take the same lock twice, causing a single-party deadlock scenario.
- Do no hold module private locks, acquired in `put` or `service` procedures, across the calls to `qprocson(9)` or `qprocsoff(9)`. These utilities spin waiting for all `put` and `service` procedures to exit, causing a single-party deadlock scenario.
- Do not hold locks, acquired in the `timeout(9)` or `bufcall(9)` callback functions across calls to `untimeout(9)` or `unbufcall(9)`. These utilities spin waiting for the callback function to exit, causing a single-party deadlock scenario.

13.11.6 MP Asynchronous Callbacks

Interrupt service routines and other asynchronous callback functions require special care by the STREAMS driver writer, because they can execute asynchronous to threads executing within the STREAMS framework.

MP-SAFE modules, or modules using synchronization barriers can use the `qtimeout(9)` and `qbufcall(9)` callbacks that are synchronous with respect to the STREAMS framework. Under *OpenSS7*, even `timeout(9)` and `bufcall(9)` utilities are synchronous with respect to the STREAMS framework when invoked from within a *qi_putp* procedure, *qi_srvp* procedure, or a synchronous callback. However, when invoked from outside a STREAMS module procedure (or from within *qi_qopen* or *qi_qclose* procedures, these functions generate asynchronous callbacks.

Because an asynchronous thread from outside of STREAMScan enter the driver at any time, the driver writer is responsible for ensuring that the asynchronous callback function acquires the necessary private locks before accessing private module data structures and releases those locks before returning. It is also the responsibility of the module to cancel any outstanding callback functions (see `untimeout(9)` and `unbufcall(9)`) before the data structures upon which they depend are deallocated and the module closed.

The following guidelines must be followed:

- Interrupt service routines must be disabled by the callback if the interrupt service routine is accessing shared data structures with the callback function.
- Outstanding callbacks from `timeout(9)` and `bufcall(9)` must be cancelled with a call to `untimeout(9)` or `unbufcall(9)`.
- Outstanding callbacks from `esballoc(9)`, must be allowed to complete before the kernel module is permitted to be unloaded.

[4] It is only true for Linux 2.4 kernels that it is necessary for the module to keep track of these things. Under recent Linux 2.6 and 3.x kernels, it is possible for the STREAMS executive to determine the module owner of the callback function and *Linux Fast-STREAMS* performs the necessary module reference counting.

13.12 Stream Integrity

The *q_next* field of the `queue(9)` structure can be dereferenced in that queue's *qi_qopen*, *qi_qclose*, *qi_putp*, and *qi_srvp* procedures as well as within any other synchronous procedure or callback (such as `qtimeout(9)`, `qbufcall(9)`, `qwriter(9)`) predicated on a queue in the same *Stream*.

All code executing outside the STREAMS framework, such as interrupt service routines, tasklets, network bottom halves, asynchronous `timeout(9)`, `bufcall(9)`, and `esballoc(9)` callback routines, are not permitted to dereference *q_next* for any queue pair in any *Stream*. Asynchronous procedures must use the '`next`' version of all functions (e.g, '`canputnext(q)`' instead of '`canput(q->q_next)`').

14 Reference

14.1 Files

14.2 System Modules

14.3 System Drivers

14.4 System Calls

14.5 Input-Output Controls

14.6 Module Entry Points

14.7 Structures

14.8 Registration

14.9 Message Handling

14.10 Queue Handling

14.11 Miscellaneous Functions

14.12 Extensions

14.13 Compatibility

15 Conformance

15.1 SVR 4.2 Compatibility

15.2 AIX Compatibility

15.3 HP-UX Compatibility

15.4 OSF/1 Compatibility

15.5 UnixWare Compatibility

15.6 Solaris Compatibility

15.7 SUX Compatibility

15.8 UXP Compatibility

16 Portability

16.1 Core Function Support

16.1.1 Core Message Functions

adjmsg(9)	trim bytes from the front or back of a STREAMS message
allocb(9)	allocate a STREAMS message and data block
bufcall(9)	install a buffer callback
copyb(9)	copy a STREAMS message block
copymsg(9)	copy a STREAMS message
datamsg(9)	tests a STREAMS message type for data
dupb(9)	duplicate a STREAMS message block
dupmsg(9)	duplicate a STREAMS message
esballoc(9)	allocate a STREAMS message and data block with a caller supplied data buffer
freeb(9)	frees a STREAMS message block
freemsg(9)	frees a STREAMS message
linkb(9)	link a message block to a STREAMS message
msgdsize(9)	calculate the size of the data in a STREAMS message
msgpullup(9)	pull up bytes in a STREAMS message
pcmsg(9)	test a data block message type for priority control
pullupmsg(9)	pull up the bytes in a STREAMS message
rmvb(9)	remove a message block from a STREAMS message
testb(9)	test if a STREAMS message can be allocated
unbufcall(9)	remove a STREAMS buffer callback
unlinkb(9)	unlink a message block from a STREAMS message

16.1.2 Core UP Queue Functions

backq(9)	find the upstream or downstream queue
bcanput(9)	test flow control on a STREAMS message queue
canenable(9)	test whether a STREAMS message queue can be scheduled
enableok(9)	allow a STREAMS message queue to be scheduled
flushband(9)	flushes band STREAMS messages from a message queue
flushq(9)	flushes messages from a STREAMS message queue
getq(9)	gets a message from a STREAMS message queue
insq(9)	inserts a message into a STREAMS message queue
noenable(9)	disable a STREAMS message queue from being scheduled
OTHERQ(9)	return the other queue of a STREAMS queue pair
putbq(9)	put a message back on a STREAMS message queue
putctl(9)	put a control message on a STREAMS message queue
putctl1(9)	put a 1 byte control message on a STREAMS message queue
putq(9)	put a message on a STREAMS message queue
qenable(9)	schedules a STREAMS message queue service routine
qreply(9)	replies to a message from a STREAMS message queue
qsize(9)	return the number of message on a queue
RD(9)	return the read queue of a STREAMS queue pair
rmvq(9)	remove a message from a STREAMS message queue

SAMESTR(9)	test for STREAMS pipe or *FIFO*
WR(9)	return the write queue of a STREAMS queue pair

16.1.3 Core MP Queue Functions

canputnext(9)	test flow control on a message queue
canputnext(9)	test flow control on a message queue
freezestr(9)	freeze the state of a stream queue
put(9s)	invoke the put procedure for a STREAMS module or driver with a STREAMS message
putnext(9)	put a message on the downstream STREAMS message queue
putnextctl1(9)	put a 1 byte control message on the downstream STREAMS message queue
putnextctl(9)	put a control message on the downstream STREAMS message queue
qprocsoff(9)	disables STREAMS message queue processing for multi-processing
qprocson(9)	enables STREAMS message queue processing for multi-processing
strqget(9)	gets information about a STREAMS message queue
strqset(9)	sets attributes of a STREAMS message queue
unfreezestr(9)	thaw the state of a stream queue

16.1.4 Core DDI/DKI Functions

kmem_alloc(9)	allocate kernel memory
kmem_free(9)	deallocates kernel memory
kmem_zalloc(9)	allocate and zero kernel memory
cmn_err(9)	print a kernel command error
bcopy(9)	copy byte strings
bzero(9)	zero a byte string
copyin(9)	copy user data in from user space to kernel space
copyout(9)	copy user data in from kernel space to user space
delay(9)	postpone the calling process for a number of clock ticks
drv_getparm(9)	driver retrieve kernel parameter
drv_hztomsec(9)	convert kernel tick time between microseconds or milliseconds
drv_htztousec(9)	convert kernel tick time between microseconds or milliseconds
drv_msectohz(9)	convert kernel tick time between microseconds or milliseconds
drv_priv(9)	check if the current process is privileged
drv_usectohz(9)	convert kernel tick time between microseconds or milliseconds
drv_usecwait(9)	delay for a number of microseconds
min(9)	determine the minimum of two integers
max(9)	determine the maximum of two integers
getmajor(9)	get the internal major device number for a device
getminor(9)	get the extended minor device number for a device
makedevice(9)	create a device from a major and minor device numbers
strlog(9)	pass a message to the STREAMS logger
timeout(9)	start a timer
untimeout(9)	stop a timer
mknod(9)	make block or character special files
mount(9)	mount and unmount file systems
umount(9)	mount and unmount file systems
unlink(9)	remove a file

16.1.5 Some Common Extension Functions

`linkmsg(9)`	link a message block to a STREAMS message
`putctl2(9)`	put a two byte control message on a STREAMS message queue
`putnextctl2(9)`	put a two byte control message on the downstream STREAMS message queue
`weldq(9)`	weld two (or four) queues together
`unweldq(9)`	unweld two (or four) queues

16.1.6 Some Internal Functions

`allocq(9)`	allocate a STREAMS queue pair
`bcanget(9)`	test for message arrival on a band on a stream
`canget(9)`	test for message arrival on a stream
`freeq(9)`	deallocate a STREAMS queue pair
`qattach(9)`	attach a module onto a STREAMS file
`qclose(9)`	close a STREAMS module or driver
`qdetach(9)`	detach a module from a STREAMS file
`qopen(9)`	call a STREAMS module or driver open routine
`setq(9)`	set sizes and procedures associated with a STREAMS message queue

16.1.7 Some Oddball Functions

`appq(9)`	append one STREAMS message after another
`esbbcall(9)`	install a buffer callback for an extended STREAMS message block
`isdatablk(9)`	test a STREAMS data block for data type
`isdatamsg(9)`	test a STREAMS data block for data type
`kmem_zalloc_node(9)`	allocate and zero memory on a node
`msgsize(9)`	calculate the size of the message blocks in a STREAMS message
`qcountstrm(9)`	add all counts on all STREAMS message queues in a stream
`xmsgsize(9)`	calculate the size of message blocks in a STREAMS message

16.2 SVR 4.2 Portability

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.

16.2.1 Differences from SVR 4.2 MP

OpenSS7 has very few differences from *SVR 4.2 MP*. Not all *SVR 4.2 MP* functions are implemented in the base *OpenSS7* kernel modules. Some functions are included in the *SVR 4.2 MP* compatibility module, `streams-svr4compat.o`.

16.2.2 Commonalities with SVR 4.2 MP

16.2.3 Compatibility functions for SVR 4.2 MP

`itimeout(9)`	Perform a timeout at an interrupt level.
`lbolt(9)`	Time in ticks since reboot.
`sleep(9)`	Put a process to sleep.
`wakeup(9)`	Wake a process.
`vtop(9)`	Convert virtual to physical address.

16.2.3.1 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 or 3.x) 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.

`spl0(9)`	Set priority level 0.
`spl1(9)`	Set priority level 1.
`spl2(9)`	Set priority level 2.
`spl3(9)`	Set priority level 3.
`spl4(9)`	Set priority level 4.
`spl5(9)`	Set priority level 5.
`spl7(9)`	Set priority level 6.
`spl7(9)`	Set priority level 7.
`spl(9)`	Set priority level.
`splx(9)`	Set priority level x.

16.2.3.2 Atomic Integers

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 and integer value from an atomic integer.
ATOMIC_INT_WRITE(9)	Write an integer value to an atomic integer.

16.2.3.3 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 head by the caller.
TRYLOCK(9)	Try to lock a basic lock.
UNLOCK(9)	Unlock a basic lock.

16.2.3.4 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.

16.2.3.5 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.

16.2.3.6 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.

16.2.3.7 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.

16.2.3.8 Resource Allocation

rmalloc(9)	Allocate a number of units from a resource map.
rmallocmap(9)	Allocated a resource map.
rmallocmap_wait(9)	Allocated a resource map.
rmalloc_wait(9)	Allocate a number of units from a resource map.
rmfree(9)	Free a number of units from a resource map.
rmfreemap(9)	Free a resource map.
rmget(9)	Allocated a number of units from a resource map.
rminit(9)	Initialize a resource map.
rmsetwant(9)	Wait for resources on a resource map.
rmwanted(9)	Waiters on a resource map.

16.2.3.9 Device Numbering

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.

16.2.4 Configuration ala SVR 4.2 MP

16.3 AIX Portability

16.3.1 Differences from AIX 5L Version 5.1

16.3.2 Commonalities with AIX 5L Version 5.1

16.3.3 Compatibility Functions for AIX 5L Version 5.1

16.3.3.1 Core Extensions

putctl2(9)	Put a 2 byte control message on a STREAMS message queue. putctl2(9) is a *OpenSS7* core function.
splstr(9)	Set or restore priority levels. splstr(9) is a *OpenSS7* core function.
splx(9)	Set or restore priority levels. splx(9) is a *OpenSS7* core function.
weldq(9)	Weld together two pairs of STREAMS message queues. weldq(9) is a *OpenSS7* core function.
unweldq(9)	Unweld two pairs of STREAMS message queues. unweldq(9) is a *OpenSS7* core function.

16.3.3.2 Common Module Utilities

mi_bufcall(9)	Reliable alternative to buffcall(9).

`mi_close_comm(9)`	STREAMS common minor device close utility.
`mi_next_ptr(9)`	STREAMS minor device list traversal.
`mi_open_comm(9)`	STREAMS common minor device open utility.
`mi_prev_ptr(9)`	STREAMS minor device list traversal.

16.3.3.3 Registration

`str_install(9)`	Install a STREAMS module or driver.

16.3.3.4 Message Filtering

`wantio(9)`	Perform direct I/O from a STREAMS driver.
`wantmsg(9)`	Provide a filter of wanted messages from a STREAMS module.

16.3.4 Configuration ala AIX 5L Version 5.1

16.4 HP-UX Portability

16.4.1 Differences from HP-UX 11.0i v2

16.4.2 Commonalities with HP-UX 11.0i v2

16.4.3 Compatibility Functions for HP-UX 11.0i v2

16.4.3.1 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(9s)`. `put(9s)` is a *OpenSS7* core function.
`putctl2(9)`	Put a 2 byte control message on a STREAMS message queue. `putctl2(9)` is a *OpenSS7* core function.
`putnextctl2(9)`	Put a 2 byte control message on the downstream STREAMS message queue. `putnextctl2(9)` is a *OpenSS7* core function.
`unweldq(9)`	Unweld two pairs of streams queues. `unweldq(9)` is a *OpenSS7* core function.
`weldq(9)`	Weld together two pairs of streams queues. `weldq(9)` is a *OpenSS7* core function.

16.4.3.2 Registration

`str_install(9)`	Install a STREAMS module or driver.
`str_uninstall(9)`	Uninstall a STREAMS module or driver.

16.4.3.3 Sleeping

`streams_get_sleep_lock(9)`	Provide access to the global sleep lock.

16.4.4 Configuration ala HP-UX 11.0i v2

16.5 OSF/1 Portability

16.5.1 Differences from OSF/1 1.2/Digital UNIX

16.5.2 Commonalities with OSF/1 1.2/Digital UNIX

16.5.3 Compatibility Functions for OSF/1 1.2/Digital UNIX

16.5.3.1 Core Extensions

`lbolt(9)` Time in ticks since reboot `lbolt(9)` is a *OpenSS7* core function.

`puthere(9)` Invoke the put procedure for a STREAMS module or driver with a STREAMS message. `puthere(9)` is implemented using `put(9s)`. `put(9s)` is a *OpenSS7* core function.

`weldq(9)` Weld together two pairs of streams queues. `weldq(9)` is a *OpenSS7* core function.

`unweldq(9)` Unweld two pairs of streams queues. `unweldq(9)` is a *OpenSS7* core function.

16.5.3.2 Common Module Utilities

`streams_close_comm(9)` Common minor device close utility.

`streams_open_comm(9)` Common minor device open utility.

`streams_open_ocomm(9)` Common minor device open utility.

16.5.3.3 Registration

`strmod_add(9)` Add a STREAMS module.

`strmod_del(9)` Delete a STREAMS module or driver from the kernel.

16.5.3.4 Others

`time(9)` (undoc).

16.5.4 Configuration ala OSF/1 1.2/Digital UNIX

16.6 UnixWare Portability

16.6.1 Differences from UnixWare 7.1.3 (OpenUnix 8)

16.6.2 Commonalities with UnixWare 7.1.3 (OpenUnix 8)

UnixWare provides most of the core functions provide by *OpenSS7* 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.

16.6.3 Compatibility Functions for UnixWare 7.1.3 (OpenUnix 8)

The following compatibility functions are in addition to all *SVR 4.2* compatibility functions.

16.6.3.1 Device Numbering

Device numbering has evolved since *UNIX Sytem 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 *OpenSS7* implementation.

`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.

16.6.3.2 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.

OpenSS7 provides these functions for compatibility, however, most of the physical requirements provided are ignored.

`allocb_physreq(9)`	Allocate a STREAMS message and data block.
`msgphysreq(9)`	Cause a message block to meet physical requirements.
`msgpullup_physreq(9)`	Pull up bytes in a STREAMS message.
`msgscgth(9)`	(undoc).

16.6.3.3 Direct STREAMS Input-Output Controls

`strioccall(9)`	(undoc).

16.6.4 Configuration ala UnixWare 7.1.3 (OpenUnix 8)

16.7 Solaris Portability

16.7.1 Differences from Solaris 9/SunOS 5.9

16.7.2 Commonalities with Solaris 9/SunOS 5.9

16.7.3 Compatibility Functions for Solaris 9/SunOS 5.9

16.7.3.1 STREAMS Queue Referenced Callbacks

`qbufcall(9)`	Install a STREAMS buffer callback.
`qunbufcall(9)`	Cancel a STREAMS buffer callback.

qtimeout(9)	Start a timer associated with a queue.
quntimeout(9)	Stop a timer associated with a queue.
qwait(9)	Wait for a queue message.
qwait_sig(9)	Wait for a queue message or signal.
queclass(9)	Return the class of a STREAMS message.
qwriter(9)	STREAMS mutex upgrade.

16.7.3.2 STREAMS Registration

install_driver(9)	Install a device driver.
mod_info(9)	Provides information on a loadable kernel module to the STREAMS executive.
mod_install(9)	Installs a loadable kernel module in the STREAMS executive.
mod_remove(9)	Removes a loadable module from the STREAMS executive.

16.7.3.3 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.

ddi_create_minor_node(9)	Create a minor node for this device.
ddi_remove_minor_node(9)	Remove a minor node for a device.
ddi_driver_major(9)	Find the major device number associated with a driver.
ddi_getiminor(9)	Get the internal minor device number.
ddi_driver_name(9)	Return normalized driver name.
ddi_get_cred(9)	Get a reference to the credentials of the current user.
ddi_get_instance(9)	Get device instance number.
ddi_get_lbolt(9)	Get the current value of the system tick clock.
ddi_get_pid(9)	Get the process id of the current process.
ddi_get_time(9)	Get the current time in seconds since the epoch.
ddi_removing_power(9)	
ddi_get_soft_state(9)	
ddi_soft_state(9)	
ddi_soft_state_fini(9)	
ddi_soft_state_free(9)	
ddi_soft_state_init(9)	
ddi_soft_state_zalloc(9)	
ddi_umem_alloc(9)	Allocate page aligned kernel memory.
ddi_umem_free(9)	Free page aligned kernel memory.

16.7.3.4 Loadable Module Interface

_fini(9)	
_info(9)	
_init(9)	
attach(9)	Attach a device to the system or resume a suspended device.
getinfo(9)	
identify(9)	Determine if a driver is associated with a device.
detach(9)	Detach a device from the system or suspend a device.

power(9) Power a device attached to the system.
probe(9)

16.7.4 Configuration ala Solaris 9/SunOS 5.9

16.8 SUX Portability

16.8.1 Differences from Super/UX

16.8.2 Commonalities with Super/UX

16.8.3 Compatibility Functions for Super/UX

lbolt(9) time in ticks since reboot

16.8.4 Configuration ala Super/UX

16.9 UXP Portability

16.9.1 Differences from UXP/V

16.9.2 Commonalities with UXP/V

16.9.3 Compatibility Functions for UXP/V

16.9.4 Configuration ala UXP/V

16.9.4.1 Extensions

lis_appq(9) Append one STREAMS message after another.
lis_date(9)
lis_esbbcall(9) Install a buffer callback for an extended STREAMS message block.
lis_find_strdev(9)
lis_OTHER(9) Return the other queue of a STREAMS queue pair.. This function is intended to accommodate a common miss-spelling of OTHERQ(9).
lis_version(9)
lis_xmsgsize(9) Calculate the size of message blocks in a STREAMS message.

16.9.4.2 Device Creation and Deletion

lis_mknod(9) Make block or character special files.
lis_unlink(9) Remove a file.
lis_mount(9) Mount a file system.
lis_umount2(9) Unmount a file system.
lis_umount(9) Unmount a file system.

16.9.4.3 Registration

`lis_register_strdev(9)`	Register a STREAMS device.
`lis_register_strmod(9)`	Register a STREAMS module.
`lis_unregister_strdev(9)`	Unregister a STREAMS device.
`lis_unregsiter_strmod(9)`	Unregister a STREAMS module.

17 Developing Portable STREAMS Modules

In the process of creating the *OpenSS7* 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.

17.1 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)`

17.2 Alignment of Message Buffers

17.3 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)`).

17.4 Freezing and Unfreezing Streams

17.5 Passing Messages from Interrupt Service Routines

17.6 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 *OpenSS7*, 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 loose the precision of the identifier.

17.7 Synchronization with Timeouts and Callback Functions

17.8 Synchronization with Callout Functions

17.9 Synchronization of Drivers and Modules

17.10 Special STREAMS Message Types

17.11 Use of Message Allocation Priorities

17.12 Registration and Deregistration

17.13 Device Numbering

17.13.1 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 ofter referred to as the "expanded device type", since it allows many more minor devices than before.

Many drivers were written for earlier releases, an 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(9)` and so on, give a machine independent interface to the device number mapping.[1]

17.13.2 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 and 3.x 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.

17.13.3 OpenSS7 Device Numbering

OpenSS7 began with extended device numbering. The `specfs` shadow special character device file system used by *OpenSS7* 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*

In earlier versions of *OpenSS7*, 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 and 3.x **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 exists 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).

`etoimajor(9)`	change external to internal major device number
`getemajor(9)`	get external major device number
`geteminor(9)`	get external minor device number
`itoemajor(9)`	change internal to external major device number

Appendix A Data Structures

A.1 Stream Structures

A.2 Queue Structures

A.3 Message Structures

A.4 Input Output Control Structures

A.5 Link Structures

A.6 Options Structures

Appendix B Message Types

B.1 Message Type

B.2 Ordinary Messages

B.3 High Priority Messages

Appendix C Utilities

Appendix D Debugging

Appendix E Configuration

Appendix F Administration

F.1 Administrative Utilities

F.2 System Controls

F.3 /proc File System

Appendix G Examples

G.1 Module Example

G.2 Driver Example

Appendix H Copying

H.1 GNU Affero General Public License

The GNU Affero General Public License.
Version 3, 19 November 2007

Copyright © 2007 Free Software Foundation, Inc. http://fsf.org/

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

H.1.1 Preamble

The GNU Affero General Public License is a free, copyleft license for software and other kinds of works, specifically designed to ensure cooperation with the community in the case of network server software.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, our General Public Licenses are intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users.

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 them 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.

Developers that use our General Public Licenses protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License which gives you legal permission to copy, distribute and/or modify the software.

A secondary benefit of defending all users' freedom is that improvements made in alternate versions of the program, if they receive widespread use, become available for other developers to incorporate. Many developers of free software are heartened and encouraged by the resulting cooperation. However, in the case of software used on network servers, this result may fail to come about. The GNU General Public License permits making a modified version and letting the public access it on a server without ever releasing its source code to the public.

The GNU Affero General Public License is designed specifically to ensure that, in such cases, the modified source code becomes available to the community. It requires the operator of a network server to provide the source code of the modified version running there to the users of that server. Therefore, public use of a modified version, on a publicly accessible server, gives the public access to the source code of the modified version.

An older license, called the Affero General Public License and published by Affero, was designed to accomplish similar goals. This is a different license, not a version of the Affero GPL, but Affero has released a new version of the Affero GPL which permits relicensing under this license.

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

Terms and Conditions

0. Definitions.

   "This License" refers to version 3 of the GNU Affero General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

   A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

   The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

   The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

   The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

   The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

   All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

   You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

   Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

   No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

   When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey 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; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

   You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   a. The work must carry prominent notices stating that you modified it, and giving a relevant date.
   b. The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".
   c. You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

d. If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.
b. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.
c. Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.
d. Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.
e. Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of

that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

   "Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

   When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

   Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

   a. Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

   b. Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

   c. Prohibiting misrepresentation of the origin of that material, or requiring that modified

versions of such material be marked in reasonable ways as different from the original version; or

d. Limiting the use for publicity purposes of names of licensors or authors of the material; or
e. Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or
f. Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

   You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

   You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

    Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

    An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

    You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

    A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

    A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

    Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

    In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

    If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

    If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a

specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

    If 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 convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Remote Network Interaction; Use with the GNU General Public License.

    Notwithstanding any other provision of this License, if you modify the Program, your modified version must prominently offer all users interacting with it remotely through a network (if your version supports such interaction) an opportunity to receive the Corresponding Source of your version by providing access to the Corresponding Source from a network server at no charge, through some standard or customary means of facilitating copying of software. This Corresponding Source shall include the Corresponding Source for any work covered by version 3 of the GNU General Public License that is incorporated pursuant to the following paragraph.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the work with which it is combined will remain governed by version 3 of the GNU General Public License.

14. Revised Versions of this License.

    The Free Software Foundation may publish revised and/or new versions of the GNU Affero 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 that a certain numbered version of the GNU Affero General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU Affero General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU Affero General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

    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.

16. Limitation of Liability.

    IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS 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.

17. Interpretation of Sections 15 and 16.

    If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

H.1.2 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 state 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 a brief idea of what it does.
Copyright (C) year name of author

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU Affero General Public License as published by
the Free Software Foundation, either version 3 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
Affero General Public License for more details.

You should have received a copy of the GNU Affero General Public License
along with this program.  If not, see http://www.gnu.org/licenses/.
```

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

If your software can interact with users remotely through a network, you should also make sure that it provides a way for users to get its source. For example, if your program is a web application, its interface could display a "Source" link that leads users to an archive of the code. There are many ways you could offer source, and different solutions will be better for different programs; see section 13 for the specific requirements.

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU AGPL, see http://www.gnu.org/licenses/.

H.2 GNU Free Documentation License

GNU FREE DOCUMENTATION LICENSE
Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.
http://fsf.org/

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

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other functional and useful 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. Secondarily, 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.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

   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. (Thus, 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. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

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 Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document 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, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. 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, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include 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, PostScript or PDF 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.

The “publisher” means any person or entity that distributes copies of the Document to the public.

A section “Entitled XYZ” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “Acknowledgements”, “Dedications”, “Endorsements”, or “History”.) To “Preserve the Title” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

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 (or copies in media that commonly have printed covers) 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 computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. 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 fewer than five), unless they release you from this requirement.

   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", Preserve 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. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements 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 to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

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, and that you preserve all their Warranty Disclaimers.

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 "Acknowledgements", 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, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which 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 half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket 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, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

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. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: 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 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ''GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

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

Glossary

anchor
A STREAMS locking mechanism that prevents the removal of STREAMS modules with the `I_POP` `ioctl`. Anchors are placed on STREAMS modules by adding the '`[anchor]`' flag to `autopush(8)` configuration files or directly with the `I_ANCHOR` `ioctl`.

autopush
A STREAMS mechanism that enables a pre-specified list of modules to be pushed automatically onto a *Stream* when a STREAMS device is opened. This mechanism is used only for administrative purposes.

back-enable
To enable (by STREAMS) a preceding blocked queue's `service` procedure when STREAMS determines that a succeeding queue has reached its low-water mark.

blocked
A queue's `service` procedure that cannot be enabled due to flow control.

clone device
A STREAMS device that returns an unused major/minor device number when initially opened, rather than requiring the minor device to be specified by name in the `open` call.

close procedure
A routine that is called when a module is popped from a *Stream* or when a driver is closed. A pointer to this procedure is specified in the *qi_qopen* member of the `queue(9)` structure associated with the read side of the module's queue pair.

control Stream
A *Stream* above a multiplexing driver used to establish lower multiplexer connections. Multiplexed *Stream* configurations are maintained through the controlling *Stream* to a multiplexing driver.

Device Driver Interface
An interface that facilitates driver portability across different *UNIX* system versions.

device driver
A *Stream* component whose principle functions are handling an associated physical device and transforming data and information between the external interface and the *Stream*.

Driver Kernel Interface
An interface between the *UNIX* system kernel and different types of drivers. It consists of a set of driver defined functions that are called by the kernel. These functions are entry points into a driver.

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

driver
A module that forms the *Stream end*. It can be a device driver or a pseudo-device driver. It is a required component in STREAMS (except in STREAMS-based pipes and FIFOs), and is physically identical to a module. It typically handles data transfer between the kernel and a device and does little or no processing of data.

enable
A term used to describe scheduling of a queue's `service` procedure.

FIFO
First In, First Out. A term used in STREAMS for named pipes. This term is also used in queue scheduling.

flow control
: A STREAMS mechanism that regulates the rate of message transfer within a *Strema* and from user space into a *Stream*.

hardware emulation module
: A module required when the terminal line discipline is on a *Stream* but there is no terminal driver at the *Stream end*. This module recognizes all `termio(7)` `ioctl`s necessary to support terminal semantics specified by `termio(9)` and `termios(9)`.

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

line discipline
: A STREAMS module that performs `termio(7)` canonical and non-canonical processing. It shares some `termio(7)` processing with a driver in a STREAMS terminal subsystem.

lower Stream
: A *Stream* connected beneath a multiplexing pseudo-device driver, by means of an `I_LINK` or `I_PLINK` `ioctl`. The far end of a lower *Stream* terminates at a device driver or another multiplexer driver.

master driver
: A STREAMS-based device supported by the pseudo-terminal subsystem. It is the controlling part of the pseudo-terminal subsystem (also called '`ptm`').

message
: One or more linked message blocks. A message is referenced by its first message block and its type is defined by the message type of that block.

message block
: A triplet consisting of a data buffer and associated control structures, a `msgb(9)` structure, a `datab(9)` structure. It carries data or information, as identified by its message type, in a *Stream*.

message queue
: A linked list of zero or more messages connected together.

message type
: A enumerated set of values identifying the contents of a message.

module
: A defined set of kernel-level routines and data structure used to process data, status, and control information on a *Stream*. It is an optional element, but there can be many modules in one *Stream*. It consists of a pair of queues (read queue and write queue), and it communicates to other components in a *Stream* by passing messages.

multiplexer
: A STREAMS mechanism that allows message to be routed among multiple *Streams* in the kernel. A multiplexing configuration includes at least one multiplexing pseudo-device driver connected to one or more upper *Streams* and one or more lower *Streams*.

named Stream
: A *Stream*, typically a pipe, with a name associated with it by way of a call to `fattach(3)` (that is, a `mount(2)` operation). This is different from a named pipe (FIFO) in two ways: a named pipe (FIFO) is unidirectional while a named *Stream* is bidirectional; a name *Stream* need not refer to a pipe, but can be another type of *Stream*.

open routine
: A procedure in each STREAMS driver and module called by STREAMS on each `open` system call made on the *Stream*. A module's `open` procedure is also called when the module is pushed.

packet mode
: A feature supported by the STREAMS-based pseudo-terminal subsystem. It is used to inform a process on the master side when state changes occur on the slave side of a pseudo-TTY. It is enabled by pushing a module called '`pckt`' on the master side.

persistent link
: A connection below a multiplexer that can exist without having an open controlling *Stream* associated with it.

pipe
: See STREAMS-based pipe.

pop
: A term used when a module that is immediately below the *Stream* head is removed.

pseudo-device driver
: A software driver, not directly associated with a physical device, that performs functions internal to a *Stream* such as a multiplexer or `log(4)` driver.

pseudo-terminal subsystem
: A user interface identical to a terminal subsystem except that there is a process in place of a hardware device. It consists of at least a master device, slave device, line discipline module, and hardware emulation module.

push
: A term used when a module is inserted in a *Stream* immediately below the *Stream head*.

pushable module
: A module put between the *Stream head* and driver. It performs intermediate transformations on messages flowing between the *Stream head* and driver. A driver is a non-pushable module.

put procedure
: A routine in a module or driver associated with a queue that receives messages from the preceding queue. It is the single entry point into a queue from a preceding queue. It may perform processing on the message and will then generally either queue the message for subsequent processing by this queue's `service` procedure, or will pass the message to the `put` procedure of the following queue (using `putnext(9)`).

queue
: A data structure that contains status information, a pointer to routines processing message, and pointers for administering a *Stream*. It typically contains pointer to `put` and `service` procedures, a message queue, and private data.

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

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

remote mode
: A feature available with the pseudo-terminal subsystem. It is used for applications that perform the canonical and echoing functions normally done by line discipline module and TTY driver. It enables applications on the master side to turn off the canonical processing.

STREAMS Administrative Driver
A STREAMS Administrative Driver that provides an interface to the `autopush(8)` mechanism.

schedule To place a queue on the internal list of queues that will subsequently have their service procedure called by the STREAMS scheduler. STREAMS scheduling is independent of *Linux* process scheduling.

service interface
A set of primitives that define a service at the boundary between a service user and a service provider and the rules (typically represented by a state machine) for allowable sequences of the primitives across the boundary. At a *Stream*/user boundary, the primitives are typically contained in the control part of a message; within a *Stream*, in `M_PROTO` or `M_PCPROTO` message blocks.

service procedure
A module or driver routine associated with a queue that receives messages queue for it by the `put` procedure is called by the STREAMS scheduler. It may perform processing on the message and generally passes the message to the `put` procedure of the following queue.

service provider
An entity in a service interface that responds to request primitives from the service user with response and event primitives.

service user
An entity in a service interface that generates request primitives for the service provider and consumes response and event primitives.

slave driver A STREAMS-based device supported by the pseudo-terminal subsystem. It is also called '`pts`' and works with a line discipline module and hardware emulation module to provide an interface to a user process.

standard pipe
A mechanism for the unidirectional flow of data between two processes where data written by one process becomes data read by the other process.

Stream A kernel level aggregate created by connecting STREAMS components, resulting from an application of the STREAMS mechanism. The primary components are the *Stream head*, the driver (or *Stream end*), and zero or more pushable modules between the *Stream head* and driver.

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

Stream end A *Stream* component furthest from the user process that contains a driver.

Stream head
A *Stream* component closest to the user process. It provides the interface between the *Stream* and the user process.

STREAMS A kernel mechanism that provides the framework for network services and data communication. It defines interface standards for character input/output within the kernel, and between the kernel and user level. The STREAMS mechanism includes integral functions, utility routines, kernel facilities, and a set of structures.

TTY driver
: A STREAMS-based device used in a terminal subsystem.

upper stream
: A *Stream* that terminates above a multiplexing driver. The beginning of an upper *Stream* originates at the *Stream head* or another multiplexing driver.

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

water mark
: A limit value used in flow control. Each queue has a high-water mark and a low-water mark. The high-water mark value indicates the upper limit related to the number of bytes contained on the queue. When the queued character reaches its high water mark, STREAMS causes another queue that attempts to send a message to this queue to become blocked. When the characters in this queue are reduced to the low-water mark value, the other queue is unblocked by STREAMS.

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

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

Index

_

_fini(9) ... 124
_info(9) ... 124
_init(9) ... 124

A

adjmsg(9) ... 115
agencies ... 6
AIX ... 104, 127
AIX 5L Version 5.1 ... 120
AIX 5L Version 5.1, commonalities ... 120
AIX 5L Version 5.1, compatibility functions ... 120
AIX 5L Version 5.1, configuration ... 121
AIX 5L Version 5.1, differences ... 120
AIX 5L Version 5.1, portability ... 120
AIX PSE ... 120, 121
allocb(9) ... 64, 115
allocb_physreq(9) ... 123
allocq(9) ... 117
anchor ... 163
ANYBAND ... 79, 80
ANYMARK ... 80, 81
appq(9) ... 117
asynchronous callbacks ... 104
asynchronous callouts ... 104
asynchronous entry points ... 104
ATOMIC_INT_ADD(9) ... 119
ATOMIC_INT_ALLOC(9) ... 119
ATOMIC_INT_DEALLOC(9) ... 119
ATOMIC_INT_DECR(9) ... 119
ATOMIC_INT_INCR(9) ... 119
ATOMIC_INT_INIT(9) ... 119
ATOMIC_INT_READ(9) ... 119
ATOMIC_INT_SUB(9) ... 119
ATOMIC_INT_WRITE(9) ... 119
attach(9) ... 124
autopush ... 163
autopush(8) ... 163, 166

B

b_band ... 64, 66, 68, 70, 71
b_cont ... 64, 66, 67
b_datap ... 64
b_flag ... 64, 80, 81
b_next ... 64, 66, 67
b_pad1 ... 64
b_pad2 ... 64
b_prev ... 64, 66, 67
b_rptr ... 64, 67
b_wptr ... 64, 67
back-enable ... 163
backq(9) ... 115
bandinfo(9) ... 78
bcanget(9) ... 117
bcanput(9) ... 75, 78, 115
bcanputnext(9) ... 75
bcopy(9) ... 116
blocked ... 163
buf ... 69, 70
bufcall(9) .. 101, 103, 106, 107, 108, 109, 115, 127
buffcall(9) ... 120
bzero(9) ... 116

C

canenable(9) ... 115
canget(9) ... 117
canput(9) ... 75, 78
canputnext(9) ... 75, 116
cdevsw(9) ... 47
clone device ... 163
close procedure ... 163
close(2s) ... 9, 14, 19, 21
close(2s) ... 29, 42
close(2s) ... 51, 68, 106
cmn_err(9) ... 116
common extension functions ... 117
commonalities, AIX 5L Version 5.1 ... 120
commonalities, HP-UX 11.0i v2 ... 121
commonalities, OSF/1 1.2/Digital UNIX ... 122
commonalities, Solaris 9/SunOS 5.9 ... 123
commonalities, Super/UX ... 125
commonalities, SVR 4.2 MP ... 118
commonalities, UnixWare 7.1.3 (OpenUnix 8) ... 122
commonalities, UXP/V ... 125
compatibility functions, AIX 5L Version 5.1 ... 120
compatibility functions, HP-UX 11.0i v2 ... 121
compatibility functions, OSF/1 1.2/Digital UNIX ... 122
compatibility functions, Solaris 9/SunOS 5.9 .. 123
compatibility functions, Super/UX ... 125
compatibility functions, SVR 4.2 MP ... 118
compatibility functions, UnixWare 7.1.3 (OpenUnix 8) ... 122
compatibility functions, UXP/V ... 125
configuration ... 102
configuration, AIX 5L Version 5.1 ... 121
configuration, HP-UX 11.0i v2 ... 121
configuration, OSF/1 1.2/Digital UNIX ... 122
configuration, Solaris 9/SunOS 5.9 ... 125
configuration, STREAMS ... 102

configuration, Super/UX . . . 125
configuration, SVR 4.2 MP . . . 120
configuration, UnixWare 7.1.3 (OpenUnix 8) . . 123
configuration, UXP/V . . . 125
contributors . . . 3
control Stream . . . 163
copyb(9) . . . 24
copyb(9) . . . 67, 115
copyin(9) . . . 116
copymsg(9) . . . 24
copymsg(9) . . . 67, 115
copyout(9) . . . 116
core ddi/dki functions . . . 116
core message functions . . . 115
core queue functions, MP . . . 116
core queue functions, UP . . . 115
credits . . . 3
CV_WAIT(9) . . . 106, 107
CV_WAIT_SIG(9) . . . 106, 107

D

D_CLONE . . . 47
D_MP . . . 103
D_MTAPAIR . . . 102
D_MTOCEXCL . . . 105
D_MTPERMOD . . . 102
D_MTPERQ . . . 103
datab(9) . . . 22, 23
datab(9) . . . 24
datab(9) . . . 61, 63, 64, 104, 164
datamsg(9) . . . 115
db_base . . . 65, 67
db_cache . . . 66
db_class . . . 65
db_filler . . . 66
db_filler2 . . . 65
db_freep . . . 65
db_frtnp . . . 65
db_iswhat . . . 65
db_lim . . . 65, 67
db_msgaddr . . . 66
db_pad . . . 65
db_ref . . . 65, 66, 67
db_size . . . 66
db_type . . . 65, 66
db_users . . . 66
DDI/DKI . . . 101, 116, 124
ddi_create_minor_node(9) . . . 124
ddi_driver_major(9) . . . 124
ddi_driver_name(9) . . . 124
ddi_get_cred(9) . . . 124
ddi_get_instance(9) . . . 124
ddi_get_lbolt(9) . . . 124
ddi_get_pid(9) . . . 124
ddi_get_soft_state(9) . . . 124
ddi_get_time(9) . . . 124
ddi_getiminor(9) . . . 124
ddi_remove_minor_node(9) . . . 124
ddi_removing_power(9) . . . 124
ddi_soft_state(9) . . . 124
ddi_soft_state_fini(9) . . . 124
ddi_soft_state_free(9) . . . 124
ddi_soft_state_init(9) . . . 124
ddi_soft_state_zalloc(9) . . . 124
ddi_umem_alloc(9) . . . 124
ddi_umem_free(9) . . . 124
delay(9) . . . 116
DELETE . . . 55
detach(9) . . . 124
dev_t . . . 128
developing portable streams modules . . . 127
device driver . . . 163
Device Driver Interface . . . 163
differences, AIX 5L Version 5.1 . . . 120
differences, HP-UX 11.0i v2 . . . 121
differences, OSF/1 1.2/Digital UNIX . . . 122
differences, Solaris 9/SunOS 5.9 . . . 123
differences, Super/UX . . . 125
differences, SVR 4.2 MP . . . 117
differences, UnixWare 7.1.3 (OpenUnix 8) . . . 122
differences, UXP/V . . . 125
document abstract . . . 7
document audience . . . 7
document disclaimer . . . 8
document information . . . 7
document intent . . . 7
document notice . . . 7
document objective . . . 7
document revisions . . . 7
downstream . . . 163
driver . . . 163
Driver Kernel Interface . . . 163
drv_getparm(9) . . . 116
drv_htztousec(9) . . . 116
drv_hztomsec(9) . . . 116
drv_msectohz(9) . . . 116
drv_priv(9) . . . 116
drv_usectohz(9) . . . 116
drv_usecwait(9) . . . 116
dupb(9) . . . 24
dupb(9) . . . 67, 115
dupmsg(9) . . . 24
dupmsg(9) . . . 67, 115

E

EBADMSG . . . 73
EINVAL . . . 79, 80, 81
EIO . . . 79
emajor(9) . . . 123
eminor(9) . . . 123

enable 163
enableok(9) 115
ENXIO 53, 79
EPIPE 79
errno(3) 79, 80, 81
esballoc(9) .. 63, 64, 102, 107, 108, 109, 115, 123
esbbcall(9) 106, 117
esbbufcall(9) 102
ESTRPIPE 80
ETIME 55
etoimajor(9) 123, 128, 129
exit(2) 19, 21, 56

F

f_inode 46
fattach(3) 22, 51, 164
fattach(8) 51
fdetach(3) 22, 51
FIFO 163
file 47, 48
file 49, 51
flow control 164
flushband(9) 75, 78, 115
flushq(9) 75, 78, 115
FLUSHR 79
FLUSHRW 79
FLUSHW 79
FMNAMESZ 53
framework integrity, STREAMS 104
free_rtn 108
freeb(9) 115
freemsg(9) 23, 115
freeq(9) 117
freezestr(9) 103, 107, 116
freezestre(9) 107

G

getemajor(9) 123, 129
geteminor(9) 123, 129
getinfo(9) 124
getmajor(9) 116
getminor(9) 116
getmsg(2p) 70
getmsg(2s) 14, 19, 23, 41
getmsg(2s) 42
getmsg(2s) 61, 68, 70, 71
getpmsg(2p) 72
getpmsg(2s) 14, 19, 23, 41
getpmsg(2s) 42
getpmsg(2s) 61, 68, 71, 72, 78
getq(9) 75, 115

H

hardware emulation module 164
HP-UX 121
HP-UX 11.0i v2 121
HP-UX 11.0i v2, commonalities 121
HP-UX 11.0i v2, compatibility functions 121
HP-UX 11.0i v2, configuration 121
HP-UX 11.0i v2, differences 121
HP-UX 11.0i v2, portability 121

I

i_pipe 47
ic_cmd 55
ic_dp 55
ic_len 55
ic_timout 55
identify(9) 124
inode 46
inode 47, 48
inode 49, 51
input side 164
insq(9) 115
install_driver(9) 124
int 79, 81
internal functions 117
ioc_cmd 55
ioct(2) 53
ioctl 56
ioctl(2p) 41, 80, 81
ioctl(2s) 9, 14, 18, 23
ioctl(2s) 29
ioctl(2s) 41
ioctl(2s) 42
ioctl(2s) ... 50, 52, 53, 54, 55, 56, 61, 62, 72, 73, 78, 79, 80, 81
isdatablk(9) 117
isdatamsg(9) 117
itimeout(9) 118
itoemajor(9) 123, 129

K

kmem_alloc(9) 63, 116, 127
kmem_fast_alloc(9) 127
kmem_fast_free(9) 127
kmem_free(9) 116, 127
kmem_zalloc(9) 116, 127
kmem_zalloc_node(9) 117

L

LASTMARK 80, 81
lbolt(9) 118, 122, 125
len 69, 70
license, AGPL 146

license, FDL . . . 156
license, GNU Affero General Public License . . . 146
license, GNU Free Documentation License . . . 156
licensing . . . 7
line discipline . . . 164
linkb(9) . . . 115
linkmsg(9) . . . 117
lis_appq(9) . . . 125
lis_date(9) . . . 125
lis_esbbcall(9) . . . 125
lis_find_strdev(9) . . . 125
lis_mknod(9) . . . 125
lis_mount(9) . . . 125
lis_OTHER(9) . . . 125
lis_register_strdev(9) . . . 126
lis_register_strmod(9) . . . 126
lis_umount(9) . . . 125
lis_umount2(9) . . . 125
lis_unlink(9) . . . 125
lis_unregister_strdev(9) . . . 126
lis_unregsiter_strmod(9) . . . 126
lis_version(9) . . . 125
lis_xmsgsize(9) . . . 125
LOCK(9) . . . 119
LOCK_ALLOC(9) . . . 119
LOCK_DEALLOC(9) . . . 119
LOCK_OWNED(9) . . . 119
log(4) . . . 165
long . . . 76, 77
lower Stream . . . 164

M

M_BACKDONE . . . 63
M_BACKWASH . . . 62
M_BREAK . . . 62
M_COPYIN . . . 62
M_COPYOUT . . . 63
M_CTL . . . 62, 63
M_DATA . . . 18, 23, 55, 61, 62, 63, 66, 68, 69, 70, 73, 74
M_DELAY . . . 62
M_DONTPLAY . . . 63
M_ERROR . . . 62, 73, 80
M_EVENT . . . 62, 63
M_FLUSH . . . 62
M_HANGUP . . . 62, 63, 73
M_HPDATA . . . 63, 68, 70
M_IOCACK . . . 62
M_IOCDATA . . . 63
M_IOCNAK . . . 62
M_IOCTL . . . 51, 55, 62, 66
M_LETSPLAY . . . 63
M_NOTIFY . . . 63
M_PASSFP . . . 62
M_PCCTL . . . 63
M_PCEVENT . . . 63
M_PCPROTO . . . 23
M_PCPROTO . . . 26
M_PCPROTO . . . 61, 62, 68, 69, 70, 73, 75, 166
M_PCRSE . . . 63
M_PCSETOPTS . . . 63
M_PCSIG . . . 62
M_PCTTY . . . 63
M_PROTO . . . 18, 23
M_PROTO . . . 26
M_PROTO . . . 61, 62, 68, 69, 70, 73, 166
M_READ . . . 62, 73
M_RSE . . . 62
M_SETOPTS . . . 62, 63, 72, 73, 74
M_SIG . . . 62
M_START . . . 62
M_STARTI . . . 63
M_STOP . . . 62
M_STOPI . . . 63
M_TRAIL . . . 62
M_UNHANGUP . . . 63
major(9) . . . 120
makedev(9) . . . 120
makedevice(9) . . . 116
master driver . . . 164
max(9) . . . 116
maxlen . . . 69, 70
message . . . 164
message block . . . 164
message ordering . . . 105
message queue . . . 164
message type . . . 164
mi_bufcall(9) . . . 101, 104, 107, 120
mi_close_comm(9) . . . 121
mi_next_ptr(9) . . . 121
mi_open_comm(9) . . . 121
mi_prev_ptr(9) . . . 121
mi_timer(9) . . . 107
min(9) . . . 116
minor(9) . . . 120
mknod(2) . . . 19
mknod(9) . . . 116
mod_info(9) . . . 124
mod_install(9) . . . 124
mod_remove(9) . . . 124
module . . . 164
module_info . . . 44
module_info(9) . . . 43
module_init . . . 43
module_init . . . 47
module_init . . . 50
module_init(9) . . . 43, 44
module_stat . . . 43, 44
module_stat(9) . . . 43, 44
mount(2) . . . 164
mount(9) . . . 116

MPSTR_QLOCK(9) 119
MPSTR_QRELE(9) 119
MPSTR_STPLOCK(9) 119
MPSTR_STPRELE(9) 119
MSG_ANY 71
MSG_BAND 71
MSG_HIPRI 71
MSGATTEN 65
msgb(9) 22
msgb(9) 24
msgb(9) 61, 63, 64, 66, 80, 81, 104, 164
MSGCOMPRESS 65
MSGDELIM 64, 80
msgdsize(9) 115
MSGLOG 65
MSGMARK 64, 80, 81
MSGMARKNET 80
MSGMARKNEXT 65
MSGNOGET 65
MSGNOLOOP 64
MSGNOTIFY 65
MSGNOTMARKNET 80
MSGNOTMARKNEXT 65
msgphysreq(9) 123
msgpullup(9) 115
msgpullup_physreq(9) 123
msgscgth(9) 123
msgsize(9) 117
multiplexer 164

N

named Stream 164
noenable(9) 115
NULL 45, 47
NULL 59, 67, 82, 84

O

O_NDELAY 21, 51
O_NDLEAY 21
O_NONBLOCK 21, 51
oddball functions 117
open routine 165
open(2s) 9, 14, 19
open(2s) 20, 29
open(2s) 31, 32, 41
open(2s) 42, 45, 47, 48
open(2s) 49, 52, 53, 68, 69, 70, 106
organization 8
OSF/1 122
OSF/1 1.2/Digital UNIX 122
OSF/1 1.2/Digital UNIX, commonalities 122
OSF/1 1.2/Digital UNIX, compatibility functions 122
OSF/1 1.2/Digital UNIX, configuration 122
OSF/1 1.2/Digital UNIX, differences 122
OSF/1 1.2/Digital UNIX, portability 122
OTHERQ(9) 115, 125

P

packet mode 165
pcmsg(9) 115
persistent link 165
pipe 165
pipe(2s) 19
pipe(2s) 29, 42, 45
pipe(2s) 49, 69, 70
poll(2s) 14, 41
poll(2s) 42
poll(2s) 80
POLLRDBAND 41
POLLWRBAND 80
pop 165
porting, AIX 5L Version 5.1 120
porting, HP-UX 11.0i v2 121
porting, OSF/1 1.2/Digital UNIX 122
porting, Solaris 9/SunOS 5.9 123
porting, Super/UX 125
porting, SVR 4.2 MP 117
porting, UnixWare 7.1.3 (OpenUnix 8) 122
porting, UXP/V 125
power(9) 125
private_data 47
probe(9) 125
pseudo-device driver 165
pseudo-terminal subsystem 165
pullupmsg(9) 115
push 165
pushable module 165
put procedure 165
put(9s) 22, 23, 28, 57, 58, 59, 60, 61, 101, 103, 104, 116, 121, 122
putbq(9) 22, 23, 75, 101, 115
putctl(9) 57, 115
putctl1(9) 57, 115
putctl2(9) 57, 117, 120, 121
puthere(9) 122
putmsg(2p) 69
putmsg(2s) 14, 19, 23, 41
putmsg(2s) 42
putmsg(2s) 61, 68, 69, 70, 71
putnext(9) 22, 23, 27, 57, 58, 59, 61, 101, 104, 105, 108, 116, 165
putnextctl(9) 57, 116
putnextctl1(9) 57, 116
putnextctl2(9) 57, 117, 121
putpmsg(2p) 71
putpmsg(2s) 14, 19, 23, 41
putpmsg(2s) 42
putpmsg(2s) 61, 68, 70, 71, 78

putq(9) 22, 23, 57, 59, 75, 101, 115

Q

q_bandp 82
q_blocked 82
q_count 77, 82
q_first 77, 82
q_flag 77, 82
q_ftmsg 83
q_hiwat 72, 77, 82
q_init 47
q_init 50
q_last 77, 82
q_link 82
q_lock 83
q_lowat 77, 82
q_maxpsz 72, 77, 82
q_minpsz 72, 77, 82
q_msgs 82
q_nband 82
q_next 47, 48
q_next 50, 82, 104, 109
q_ptr 82, 104, 106
q_qinfo 82
q_qptr 105
qattach(9) 117
QB_BACK 84
qb_count 77, 84
qb_first 77, 84
qb_flag 77, 84
QB_FULL 84
qb_hiwat 77, 84
qb_last 77, 84
qb_lowat 77, 84
qb_maxpsz 77
qb_minpsz 77
qb_msgs 84
qb_next 84
qb_pad1 84
qb_padq 84
QB_WANTW 84
QBACK 83
qband(9) 26
qband(9) 43, 44, 75, 76, 77, 82, 84
qband_t(9) 84
qbufcall(9) 101, 104, 106, 107, 108, 109, 123
qclose(9) 101, 103, 117, 127
QCOUNT 77
qcountstrm(9) 117
qdetach(9) 117
QENAB 83
qenable(9) 115
qfields_t 77
qfields_t(9) 76
QFIRST 77
QFLAG 77
QFULL 83
QHIWAT 77
QHLIST 83
qi_lowat 72
qi_minfo 47
qi_putp 57, 58, 104, 108, 109
qi_qadmin 57
qi_qclose 50, 57, 68, 105, 106, 107, 108, 109
qi_qopen 47
qi_qopen 50, 57, 68, 106, 107, 108, 109, 163
qi_srvp 57, 59, 104, 108, 109
qinit 45, 47
qinit 50
qinit(9) 43, 44, 57, 58, 59, 82, 104
QLAST 77
QLOWAT 77
QMAXPSZ 77
QMINPSZ 77
QNOENB 83
QOLD 83
qopen(9) 101, 103, 117, 127
qpad1 82
QPROCS 83
qprocsoff(9) 103, 105, 106, 108, 116, 127
qprocson(9) 103, 106, 108, 116, 127
QREADR 83
qreply(9) .. 23, 27, 57, 61, 101, 104, 105, 108, 115
QSAFE 83
qsize(9) 115
QSVCBUSY 83
QSYNCH 83
qtimeout(9) 101, 104, 106, 107, 108, 109, 124
QTOENAB 83
queclass(9) 124
queue 46
queue 47
queue 50
queue 165
queue(9) 25
queue(9) 26
queue(9) ... 43, 44, 57, 59, 75, 76, 77, 82, 83, 104, 109, 163
qunbufcall(9) 104, 107, 123
quntimeout(9) 104, 107, 124
QUP 83
QUSE 83
qwait(9) 106, 107, 124
qwait_sig(9) 106, 107, 124
QWANTR 83
QWANTW 83
QWCLOSE 83
QWELDED 83
qwriter(9) 103, 109, 124

R

RD(9) . . . 115
read queue . . . 165
read(2s) . . . 9, 14, 19, 21, 23, 41
read(2s) . . . 42
read(2s) . . . 61, 68, 72, 73, 80, 81, 165
read-side . . . 165
readv(2s) . . . 72, 73
remote mode . . . 165
RFILL . . . 73
rmalloc(9) . . . 120, 127
rmalloc_wait(9) . . . 120, 127
rmallocmap(9) . . . 120, 127
rmallocmap_wait(9) . . . 120
rmfree(9) . . . 120, 127
rmfreemap(9) . . . 120, 127
rmget(9) . . . 120
rminit(9) . . . 120, 127
rmsetwant(9) . . . 120, 127
RMSGD . . . 72
RMSGN . . . 72
rmvb(9) . . . 115
rmvq(9) . . . 115
rmwanted(9) . . . 120, 127
RNORM . . . 72, 73
RPROCOMPRESS . . . 73
RPROTCOMPRESS . . . 73
RPROTDAT . . . 73
RPROTDIS . . . 73
RPROTNORM . . . 73
RW_ALLOC(9) . . . 119
RW_DEALLOC(9) . . . 119
RW_RDLOCK(9) . . . 119
RW_TRYRDLOCK(9) . . . 119
RW_TRYWRLOCK(9) . . . 119
RW_UNLOCK(9) . . . 119
RW_WRLOCK(9) . . . 119

S

S_BANDURG . . . 78, 81, 82
S_IFIFO . . . 49
S_RDBAND . . . 78, 81, 82
S_WRBAND . . . 78, 81, 82
SAMESTR(9) . . . 116
schedule . . . 166
sd_file . . . 47
sd_inode . . . 47
service interface . . . 166
service procedure . . . 166
service provider . . . 166
service user . . . 166
setq(9) . . . 117
sfx(4) . . . 49
SIGPIPE . . . 49, 73
SIGPOLL . . . 78, 81, 82
SIGURG . . . 78, 81, 82
slave driver . . . 166
sleep(9) . . . 118
SLEEP_ALLOC(9) . . . 119
SLEEP_DEALLOC(9) . . . 119
SLEEP_LOCK(9) . . . 119
SLEEP_LOCK_SIG(9) . . . 119
SLEEP_LOCKAVAIL(9) . . . 119
SLEEP_LOCKOWNED(9) . . . 119
SLEEP_TRYLOCK(9) . . . 119
SLEEP_UNLOCK(9) . . . 119
SNDHOLD . . . 74
SNDPIPE . . . 49, 73
SNDZERO . . . 49, 73
snode . . . 46
snode . . . 47
so_readopt . . . 72
SO_READOPT . . . 72
so_wroff . . . 74
SO_WROFF . . . 74
sockmod(4) . . . 73
Solaris . . . 123, 125
Solaris 9/SunOS 5.9 . . . 123, 125
Solaris 9/SunOS 5.9, commonalities . . . 123
Solaris 9/SunOS 5.9, compatibility functions . . . 123
Solaris 9/SunOS 5.9, configuration . . . 125
Solaris 9/SunOS 5.9, differences . . . 123
Solaris 9/SunOS 5.9, portability . . . 123
spl(9) . . . 118
spl0(9) . . . 118
spl1(9) . . . 118
spl2(9) . . . 118
spl3(9) . . . 118
spl4(9) . . . 118
spl5(9) . . . 118
spl7(9) . . . 118
splstr(9) . . . 120
splx(9) . . . 118, 120
sponsors . . . 3
SQLVL_DEFAULT . . . 102
SQLVL_ELSEWHERE . . . 102, 105
SQLVL_GLOBAL . . . 102, 105
SQLVL_MODULE . . . 102, 105
SQLVL_NOP . . . 103
SQLVL_QUEUE . . . 102, 105
SQLVL_QUEUEPAIR . . . 102, 105
srv(9s) . . . 101, 103
st_muxrinit . . . 45
st_muxwinit . . . 45
st_rdinit . . . 45
st_wrinit . . . 45
standard pipe . . . 166
stdata . . . 46
stdata . . . 47, 48
stdata . . . 49, 51
stdata(9) . . . 46

str_install(9) ... 121
str_uninstall(9) ... 121
strbuf(5) ... 69, 70
Stream ... 166
Stream end ... 166
Stream head ... 166
streamio(7) ... 11, 23, 31, 41, 56, 61, 74, 78, 127
STREAMS ... 166
STREAMS Administrative Driver ... 166
STREAMS(9) ... 7, 11
STREAMS, configuration ... 102
STREAMS, framework integrity ... 104
STREAMS-based pipe ... 166
streams_close_comm(9) ... 122
streams_get_sleep_lock(9) ... 121
streams_open_comm(9) ... 122
streams_open_ocomm(9) ... 122
streams_put(9) ... 121
streamtab ... 45, 47
streamtab ... 50
streamtab(9) ... 45
strioccall(9) ... 123
strioctl ... 54
strlog(9) ... 116
strmod_add(9) ... 122
strmod_del(9) ... 122
stroptions(9) ... 72, 74
strqget(9) ... 76, 77, 116
strqset(9) ... 76, 77, 116
SunOS ... 123, 125
Super/UX ... 125
Super/UX, commonalities ... 125
Super/UX, compatibilty functions ... 125
Super/UX, configuration ... 125
Super/UX, differences ... 125
Super/UX, portability ... 125
supporters ... 3
SV_ALLOC(9) ... 119
SV_BROADCAST(9) ... 119
SV_DEALLOC(9) ... 120
SV_SIGNAL(9) ... 120
SV_WAIT(9) ... 120
SV_WAIT_SIG(9) ... 120
SVR 4.2 ... 122
SVR 4.2 MP ... 102, 117, 118, 120, 122, 124
SVR 4.2 MP DDI/DKI ... 101
SVR 4.2 MP, commonalities ... 118
SVR 4.2 MP, compatibility functions ... 118
SVR 4.2 MP, configuration ... 120
SVR 4.2 MP, differences ... 117
SVR 4.2 MP, portability ... 117
SVR4 ... 127
synchronization, default ... 102
synchronization, elsewhere ... 102
synchronization, global ... 102
synchronization, module ... 102
synchronization, none ... 103
synchronization, queue ... 102
synchronization, queue pair ... 102
synchronous callbacks ... 104
synchronous callouts ... 104
synchronous entry points ... 103

T

tcp(4) ... 81
termio(7) ... 164
termio(9) ... 164
termios(9) ... 164
testb(9) ... 115
time(9) ... 122
timeout(9) ... 102, 103, 106, 107, 108, 109, 116, 127
timod(4) ... 73
tirdwr(4) ... 73
TRYLOCK(9) ... 119
TTY driver ... 167

U

umount(9) ... 116
unbufcall(9) ... 107, 108, 115
unfreezestr(9) ... 103, 116
UNIX System V Release 3.0 ... 13
UNIX System V Release 4 ... 13
UNIX System V Release 4.2 ... 13, 104
UnixWare ... 122, 123
UnixWare 7.1.3 (OpenUnix 8) ... 122, 123
UnixWare 7.1.3 (OpenUnix 8), commonalities ... 122
UnixWare 7.1.3 (OpenUnix 8), compatibility functions ... 122
UnixWare 7.1.3 (OpenUnix 8), configuration ... 123
UnixWare 7.1.3 (OpenUnix 8), differences ... 122
UnixWare 7.1.3 (OpenUnix 8), portability ... 122
unlink(9) ... 116
unlinkb(9) ... 115
UNLOCK(9) ... 119
untimeout(9) ... 107, 108, 116
unweldq(9) ... 117, 120, 121, 122
upper stream ... 167
upstream ... 167
UXP/V ... 125
UXP/V, commonalities ... 125
UXP/V, compatibility functions ... 125
UXP/V, configuration ... 125
UXP/V, differences ... 125
UXP/V, portability ... 125

V

vnode ... 46
vnodes ... 49

vtop(9) 118

W

wakeup(9) 118
wantio(9) 121
wantmsg(9) 121
water mark 167
weldq(9) 117, 120, 121, 122
WR(9) 116
write queue 167
write(2s) 9, 14, 19, 21, 23, 41
write(2s) 42
write(2s) 61, 68, 73, 74, 167
write-side 167

X

XCASE 55
XENIX 123
xmsgsize(9) 117
xti(3) 73