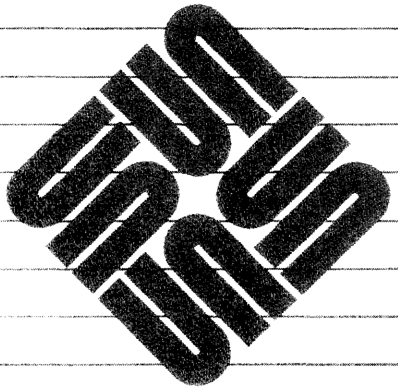
sun®
microsystems

Writing Device Drivers



Part Number: 800-1780-10
Revision A, of 9 May 1988

Sun™, **Sun-2™**, **Sun-3™**, and **Sun-4™** are trademarks of Sun Microsystems, Incorporated. **Sun Workstation®** is a registered trademark of Sun Microsystems, Inc.

Multibus is a trademark of Intel Corporation.

UNIX is a trademark of AT&T Bell Laboratories.

VMEbus is a trademark of Motorola, Incorporated.

VAX is a trademark of Digital Equipment Corporation.

IBM-PC and **IBM 370** are trademarks of International Business Machines Corporation.

Cray is a trademark of Cray Research.

All other products or services mentioned in this document are identified by the trademarks or service marks of their respective companies or organizations

Sun equipment generates, uses, and can radiate radio frequency energy and if not installed and used in accordance with the instructions manual, may cause interference to radio communications. It has been tested and found to comply with the limits for a Class A computing device pursuant to Subpart J of Part 15 of FCC Rules, which are designed to provide reasonable protection against such interference when operated in a commercial environment. Operation of Sun equipment in a residential area is likely to cause interference in which case the user at his own expense will be required to take whatever measures may be required to correct the interference.

Copyright © 1987, 1988 by Sun Microsystems, Inc.

This publication is protected by Federal Copyright Law, with all rights reserved. No part of this publication may be reproduced, stored in a retrieval system, translated, transcribed, or transmitted, in any form, or by any means manual, electric, electronic, electro-magnetic, mechanical, chemical, optical, or otherwise, without prior explicit written permission from Sun Microsystems.

Contents

Chapter 1 Introduction **3**

1.1. Device Independence 3
1.2. Types of Devices 4
1.3. System V Compatibility 6
1.4. Major Development Stages 6
1.5. Warning To Microcomputer Programmers 6
1.6. Address-Space Terminology 7
1.7. Manual Overview 8
    Regular Drivers 8
    STREAMS Drivers 8
    Last Word 8

PART ONE: Regular Device Drivers 9

Chapter 2 Hardware Context **13**

2.1. Multibus Machines 13
    Multibus Memory Address Space and I/O Address Space 13
    Allocation of Multibus Memory 16
    Allocation of Multibus I/O Space 17
2.2. VMEbus Machines 18
    Sun-2 VMEbus Address Spaces 18
    Sun-3/Sun-4 Address Spaces 20
    Allocation of VMEbus Memory 22

The Sun VMEbus to Multibus Adapter 24
Interrupt Vector Assignments 24
2.3. ATbus Machines 25
Loadable Drivers 27
DOS and SunOS Environments 27
2.4. Hardware Peculiarities to Watch Out For 28
Multibus Device Peculiarities 28
Multibus Byte-Ordering Issues 28
Other Multibus-related Peculiarities 30
Sun-4/SPARC Peculiarities 31
Other Device Peculiarities 32
2.5. DMA Devices 33
Sun Main-Bus DVMA 33
DMA on ATbus Machines 36

Chapter 3 Overall Kernel Context **41**
3.1. The System Kernel 41
3.2. Devices as ''Special'' Files 42
3.3. Run-Time Data Structures 47
The Bus-Resource Interface 49
Autoconfiguration-Related Declarations 55
Other Kernel/Driver Interfaces 56

Chapter 4 Kernel Topics and Device Drivers **61**
4.1. Overall Layout of a Character Device Driver 61
4.2. User Space versus Kernel Space 63
4.3. User Context and Interrupt Context 63
4.4. Device Interrupts 64
4.5. Interrupt Levels 65
4.6. Vectored Interrupts and Polling Interrupts 66
4.7. Some Common Service Routines 69
Timeout Mechanisms 69
Sleep and Wakeup Mechanism 69

Raising and Lowering Processor Priorities 70
Main Bus Resource Management Routines 71
Data-Transfer Functions 71
Kernel **`printf()`** Function 72
Macros to Manipulate Device Numbers 72

Chapter 5 Driver Development Topics 75
5.1. Installing and Checking the Device 75
Setting the Memory Management Unit 75
Selecting a Virtual Address 76
Finding a Physical Address 79
Selecting a Virtual to Physical Mapping 79
Sun-2 Address Mapping 81
Sun-3 and Sun-4 Address Mapping 84
A Few Example PTE Calculations 87
Getting the Device Working and in a Known State 88
A Warning about Monitor Usage 90
5.2. Installation Options for Memory-Mapped Devices 90
Memory-Mapped Device Drivers 90
Mapping Devices Without Device Drivers 92
Direct Opening of Memory Devices 95
5.3. Debugging Techniques 97
Debugging with **`printf()`** 98
Event-Triggered Printing 100
Asynchronous Tracing 101
`kadb` — A Kernel Debugger 102
5.4. Device Driver Error Handling 103
Error Recovery 103
Error Returns 103
Error Signals 104
Error Logging 104
Kernel Panics 104
5.5. System Upgrades 105

5.6. Loadable Drivers 105

Chapter 6 The "Skeleton" Character Device Driver **111**

6.1. General Declarations in Driver 114

6.2. Autoconfiguration Procedures 115

- `probe()` Routine 115
- `attach()` Routine 117

6.3. `open()` and `close()` Routines 117

6.4. `read()` and `write()` Routines 119

- Some Notes About the UIO Structure 120

6.5. Skeleton `strategy()` Routine 121

6.6. Skeleton `start()` Routine 122

6.7. `intr()` and `poll()` Routines 124

6.8. `ioctl()` Routine 126

6.9. Skeleton Driver Variations 126

- DMA Variations 126
  - Multibus or VMEbus DVMA 126
  - A DMA Skeleton Driver 127
- Variation with "Asynchronous I/O" Support 130
  - Select Routines 131
  - Adding Asynchronous Notification 134
  - Adding an `ioctl()` routine 134

Chapter 7 Configuring the Kernel **139**

7.1. Background Information 139

7.2. An Example 141

7.3. Devices that use Two Address Spaces 145

7.4. Adding and Removing Loadable Drivers 146

Chapter 8 Pseudo-Device Drivers — A Ramdisk **151**

8.1. A Ramdisk Driver 152

- Ramdisk Source Code 152
- Ramdisk Installation 153

Ramdisk Test Program 156

PART TWO: STREAMS Programming 157

Chapter 9 Introduction to STREAMS 161

9.1. A Basic View of a Stream 162
  System Calls 163
9.2. Benefits of STREAMS 165
  Creating Service Interfaces 165
  Manipulating Modules 165
  Protocol Portability 165
  Protocol Substitution 166
  Protocol Migration 166
  Module Reusability 167
9.3. An Advanced View of a Stream 168
  Stream Head 169
  Modules 169
  Stream End 170
9.4. Building a Stream 171
  Expanded Streams 172
  Pushable Modules 172
9.5. Basic User Level Functions 173
  STREAMS System Calls 173
  An Asynchronous Protocol Stream Example 174
    Initializing the Stream 175
    Message Types 176
    Sending and Receiving Messages 176
    Using Messages in the Example 177
  Other User Functions 180
9.6. Kernel Level Functions 180
  Messages 180
    Message Allocation 182

Put and Service Procedures 183
  Put Procedures 183
  Service Procedures 183
Kernel Processing 184
  Read Side Processing 185
  Driver Processing 185
    CHARPROC 185
    CANONPROC 186
  Write Side Processing 186
  Analysis 187
9.7. Other Facilities 187
  Message Queue Priority 187
  Flow Control 188
  Multiplexing 190
  Monitoring 192
  Error and Trace Logging 193
9.8. Driver Design Comparisons 195
  Environment 195
  Drivers 195
  Modules 196
9.9. Glossary 196

Chapter 10 STREAMS Applications Programming **201**
10.1. Introduction 201
  Streams Overview 201
  Development Facilities 203
10.2. Basic Operations 204
  A Simple Stream 204
  Inserting Modules 206
  Module and Driver Control 207
10.3. Advanced Operations 210
  Advanced Input/Output Facilities 210
  Input/Output Polling 210

Asynchronous Input/Output 213
Clone Open 214
10.4. Multiplexed Streams 214
Multiplexor Configurations 214
Building a Multiplexor 216
Dismantling a Multiplexor 221
Routing Data Through a Multiplexor 222
10.5. Message Handling 223
Service Interface Messages 223
Service Interfaces 223
The Message Interface 224
Datagram Service Interface Example 226
Accessing the Datagram Provider 228
Closing the Service 231
Sending a Datagram 231
Receiving a Datagram 232

Chapter 11 STREAMS Module and Driver Programming 237
11.1. Introduction 237
Development Facilities 238
11.2. Streams Mechanism 238
Stream Construction 239
Opening a Stream 241
Adding and Removing Modules 242
Closing 242
11.3. Modules 243
Module Declarations 243
Module Procedures 245
Module and Driver Environment 246
11.4. Messages 247
Message Format 247
Message Generation and Reception 249
Filter Module Declarations 249

        `bappend()` Subroutine 250
    Message Allocation 251
    Put Procedure 251
11.5. Message Queues and Service Procedures 253
    The **`queue_t`** Structure 253
    Service Procedures 254
    Message Queues and Message Priority 254
    Flow Control 255
    Example 256
        Procedures 257
11.6. Drivers 259
    Overview of Drivers 259
    Driver Flow Control 261
    Driver Programming 262
        Driver Declarations 262
        Driver Open 264
    Driver Processing Procedures 265
        Driver Flush Handling 266
        Driver Interrupt 266
    Driver and Module Ioctls 267
    Driver Close 269
11.7. Complete Driver 269
    Cloning 269
    Loop-Around Driver 270
        Write Put Procedure 273
        Stream Head Messages 276
        Service Procedures 276
        Close 277
11.8. Multiplexing 278
    Multiplexing Configurations 278
        Connecting Lower Streams 279
        Disconnecting Lower Streams 281
    Multiplexor Construction Example 281

Multiplexing Driver 284
  Upper Write Put Procedure 287
  Lower **QUEUE** Write Service Procedure 290
  Lower Read Put Procedure 292
11.9. Service Interface 294
  Definition 294
    Message Usage 294
  Example 295
    Declarations 295
    Service Interface Procedure 297
11.10. Advanced Topics 299
  Recovering From No Buffers 299
  Advanced Flow Control 301
  Signals 302
  Control of Stream Head Processing 303
    Read Options 303
    Write Offset 303

Chapter 12 SunOS STREAMS Topics **307**
12.1. Configuring STREAMS Drivers 307
  Module Configuration 308
  Tunable Parameters 309
  System Error Messages 310
12.2. STREAMS in SunOS 311
  STREAM Modules 311
  SunOS STREAMS Extension 312
  STREAMS Portability 312
  User Line Disciplines 312

Appendix A Supplementary STREAMS Material **317**
A.1. Kernel Structures 317
  `streamtab` 317
  QUEUE Structures 317

A.2. Message Structures 318

`iocblk` 319

`linkblk` 319

A.3. Message Types 320

Ordinary Messages 320

Priority Messages 325

A.4. Utilities 327

Buffer Allocation Priority 328

`adjmsg()` — Trim Bytes in a Message 329

`allocb()` — Allocate a Message Block 329

`backq()` — Get Pointer to Queue Behind a Given Queue 329

`bufcall()` — Recover from Failure of **`allocb()`** 330

`canput()` — Test for Room in a Queue 330

`copyb()` — Copy a Message Block 330

`copymsg()` — Copy a Message 331

`datamsg()` — Test Whether Message is a Data Message 331

`dupb()` — Duplicate a Message Block Descriptor 331

`dupmsg()` — Duplicate a Message 331

`enableok()` — Re-allow Queue to be Scheduled 332

`flushq()` — Flush a Queue 332

`freeb()` — Free a Message Block 332

`freemsg()` — Free All Message Blocks in a Message 332

`getq()` — Get a Message from a Queue 332

`insq()` — Put a Message at a Specific Place in a Queue 333

`linkb()` — Concatenate Two Messages into One 333

`msgdsize()` — Get Number of Data Bytes in a Message 333

`noenable()` — Prevent a Queue from Being Scheduled 333

`OTHERQ()` — Get Pointer to the Mate Queue 334

`pullupmsg()` — Concatenate Bytes in a Message 334

`putbq()` — Return a Message to the Beginning of a Queue 334

`putctl()` — Put a Control Message 334

`putctl1()` — Put One-byte Parameter Control Message 335

`putnext()` — Put a Message to the Next Queue 335

`putq()` — Put a Message on a Queue 335

`qenable()` — Enable a Queue 336

`qreply()` — Send Reverse-Direction Message on Stream 336

`qsize()` — Find the Number of Messages on a Queue 336

`RD()` — Get Pointer to the Read Queue 336

`rmvb()` — Remove a Message Block from a Message 336

`rmvq()` — Remove a Message from a Queue 337

`splstr()` — Set Processor Level 337

`strlog()` — Submit Messages for Logging 337

`testb()` — Check for an Available Buffer 337

`unlinkb()` – Remove Message Block from Message Head 338

`WR()` — Get Pointer to the Write Queue 338

A.5. Design Guidelines 338

General Rules 338

System Calls 339

Data Structures 339

Header Files 340

Accessible Symbols and Functions 340

Rules for Put and Service Procedures 341

A.6. STREAMS Glossary 343

PART THREE: Non-STREAMS Appendices 347

Appendix B Summary of Device Driver Routines **351**

B.1. Standard Error Numbers 351

B.2. Device Driver Routines 351

*xx***`attach()`** — Attach a Slave Device 352

*xx***`close()`** — Close a Device 352

*xx***`intr()`** — Handle Vectored Interrupts 352

*xx***`ioctl()`** — Miscellaneous I/O Control 353

*xx***`mmap()`** — Mmap a Page of Memory 355

*xx***`minphys()`** — Determine Maximum Block Size 355

*xx***open()** — Open a Device for Data Transfers 356
*xx***poll()** — Handle Polling Interrupts 357
*xx***probe()** — Determine if Hardware is There 357
*xx***read()** — Read Data from Device 358
*xx***select()** — Select Support 358
*xx***strategy()** — High-Level I/O 359
*xx***write()** — Write Data to Device 359

Appendix C Kernel Support Routines **363**
btodb() — Convert Bytes to Disk Sectors 363
copyin() — Move Data From User to Kernel Space 363
copyout() — Move Data From Kernel to User Space 363
CDELAY() — Conditional Busy Wait 364
DELAY() — Busy Wait for a Given Period 364
dma_done() — Free the DMA Channel 364
dma_setup() — Set Up for a DMA Transfer 364
gsignal() — Send Signal to Process Group 368
hat_getkpfnum() — Address to Page Frame Number 368
inb() — Read a Byte from an I/O Port 368
iodone() — Indicate I/O Complete 369
iowait() — Wait for I/O to Complete 369
kmem_alloc() — Allocate Space from Kernel Heap 369
kmem_free() — Return Space to Kernel Heap 369
log() — Log Kernel Errors 370
MBI_ADDR() — Get Address in DVMA Space 370
mapin() — Map Physical to Virtual Addresses 370
mapout() — Remove Physical to Virtual Mappings 372
mbrelse() — Free Main Bus Resources 372
mbsetup() — Set Up to Use Main Bus Resources 372
outb() — Send a Byte to an I/O Port 373
panic() — Reboot at Fatal Error 373
peek(), **peekc()**, **peekl()** — Check and Read 373
physio() — Block I/O Service Routine 373

poke(), pokec(), pokel() — Check and Write 375
printf() — Kernel Printf Function 376
pritospl() — Convert Priority Level 376
psignal() — Send Signal to Process 377
rmalloc() — General-Purpose Resource Allocator 377
rmfree() — Recycle Map Resource 378
selwakeup() — Wakeup a Select-blocked Process 378
sleep() — Sleep on an Event 378
spl*n*() — Set CPU Priority Level 379
splx() — Reset Priority Level 379
suser() — Reset Priority Level 380
swab() — Swap Bytes 380
timeout() — Wait for an Interval 380
uiomove() — Move Data To or From an **uio** Structure 380
untimeout() — Cancel **timeout()** Request 381
uprintf() — Nonsleeping Kernel Printf Function 381
ureadc(), uwritec() — `uio` Structure Read/Write 381
wakeup() — Wake Up a Process Sleeping on an Event 382

Appendix D User Support Routines **385**
free() — Free Allocated Memory 385
getpagesize() — Return Pagesize 385
mmap() — Map Memory from One Space to Another 385
munmap() — Unmap Pages of Memory 386

Appendix E Sample Driver Listings **389**
E.1. Skeleton Board Driver 390
E.2. Sun-2 Color Graphics Driver 398
E.3. Sky Floating-Point Driver 415
E.4. Versatec Interface Driver 423
E.5. Sun386i Parallel Port Driver 435
Index 445

Tables

Table 1-1 VMEbus Address-space Names 7

Table 2-1 Sun-2 Multibus Memory Types 14
Table 2-2 Sun-2 Multibus Memory Map 17
Table 2-3 Sun-2 Multibus I/O Map 17
Table 2-4 Sun-2 VMEbus Memory Types 18
Table 2-5 Generic VMEbus (Full Set) 20
Table 2-6 Sun-3/Sun-4 VMEbus Address Types 20
Table 2-7 16-bit VMEbus Address Space Allocation 23
Table 2-8 24-bit VMEbus Address Space Allocation 23
Table 2-9 32-bit VMEbus Address Space Allocation (Sun-3s and Sun-4s Only) 23
Table 2-10 VMEbus Address Assignments for Some Devices 24
Table 2-11 Vectored Interrupt Assignments 25
Table 2-12 Interrupt Channel Assignments 26
Table 2-13 Sun386i DMA Channel Assignments 27

Table 3-1 A Sample Listing of the /dev Directory 43
Table 3-2 Current Major Device Number Assignments 46

Table 5-1 Sun-2 PTE Masks 82
Table 5-2 Sun-3/Sun-4 PTE Masks 86
Table 5-3 Virtual Memory Devices 93

Figures

Figure 2-1 Sun-2 Multibus Address Spaces 15
Figure 2-2 Sun-2 VMEbus Address Spaces 19
Figure 2-3 Sun-3 VMEbus Address Spaces 21
Figure 2-4 Sun-4 VMEbus Address Spaces 22
Figure 2-5 System DVMA 35

Figure 3-1 I/O Paths in the UNIX system 44

Figure 5-1 Sun-2 Address Mapping 77
Figure 5-2 Sun-3 Address Mapping 77
Figure 5-3 Sun-4 Address Mapping 78
Figure 5-4 Sun386i Address Mapping 78
Figure 5-5 Sun-2 MMU 81
Figure 5-6 Sun-3 MMU 84
Figure 5-7 Sun-4 MMU 85

Figure 9-1 Basic Stream 163
Figure 9-2 STREAMS-Related Manual Pages 164
Figure 9-3 Protocol Module Portability 166
Figure 9-4 Protocol Migration 167
Figure 9-5 Module Reusability 168
Figure 9-6 Stream In More Detail 169
Figure 9-7 Setting Up a Stream 171
Figure 9-8 Idle Stream Configuration for Example 175

Figure 9-9 Asynchronous Terminal Streams 179
Figure 9-10 A Message 181
Figure 9-11 Messages on a Message Queue 182
Figure 9-12 Operational Stream for Example 184
Figure 9-13 Module Put and Service Procedures 185
Figure 9-14 Streams Message Priority 188
Figure 9-15 Flow Control 189
Figure 9-16 Internet Multiplexing Stream 190
Figure 9-17 X.25 Multiplexing Stream 191
Figure 9-18 Error and Trace Logging 194

Figure 10-1 Basic Stream 203
Figure 10-2 Stream to Communications Driver 205
Figure 10-3 Case Converter Module 207
Figure 10-4 Many-to-one Multiplexor 215
Figure 10-5 One-to-many Multiplexor 215
Figure 10-6 Many-to-many Multiplexor 215
Figure 10-7 Protocol Multiplexor 216
Figure 10-8 Before Link 217
Figure 10-9 IP Multiplexor After First Link 218
Figure 10-10 IP Multiplexor 219
Figure 10-11 TP Multiplexor 220
Figure 10-12 Protocol Substitution 223
Figure 10-13 Service Interface 224

Figure 11-1 Downstream Stream Construction 240
Figure 11-2 QUEUE data structures 240
Figure 11-3 Message Form and Linkage 248
Figure 11-4 Message Queue Priority 255
Figure 11-5 Device Driver Streams 261
Figure 11-6 Loop Around Streams 271
Figure 11-7 Internet Multiplexor Before Connecting 282
Figure 11-8 Internet Multiplexor After Connecting 283

Figure 11-9 Example Multiplexor Configuration 287

Figure A-1 `M_PROTO` and `M_PCPROTO` Message Structure 321

1

Introduction

Introduction 3

1.1. Device Independence 3
1.2. Types of Devices 4
1.3. System V Compatibility 6
1.4. Major Development Stages 6
1.5. Warning To Microcomputer Programmers 6
1.6. Address-Space Terminology 7
1.7. Manual Overview 8
    Regular Drivers 8
    STREAMS Drivers 8
    Last Word 8

1

Introduction

This manual is a guide to adding drivers for new devices to the SunOS kernel. It comes is three parts.

- Part One, *Regular Device Drivers*, discusses a variety of issues relevant to standard (non-STREAMS) device drivers. It is intended to be self-contained, and to include all necessary discussion of hardware and kernel topics.
- Part Two, *STREAMS Programming*, discusses topics relevant to the construction and installation of STREAMS drivers and modules. It also includes STREAMS-related reference material.
- Part Three, *Non-STREAMS Appendices*, includes reference material related to regular (non-STREAMS) drivers.

Throughout the manual, statements that apply only to specific machines, e.g. Sun-4s or Sun386i's, will be clearly flagged to that effect.

1.1. Device Independence

One of SunOS's major services to application programs is to provide a device-independent view of the I/O hardware. In this view, user processes (application programs), see devices as "special" types of files that can be opened, closed and manipulated just like regular files. The user process manipulates devices as it would files, by making *system calls*.

Once a system call carries process execution into the SunOS kernel, however, it becomes clear just how "special" devices really are. The kernel distinguishes between real files and device special files, and translates operations on the latter into calls to their corresponding device drivers. These drivers control *all* device operations; devices do nothing until their drivers tell them to.

Thus, system calls provide the interface between user processes and the SunOS kernel, while device drivers provide an interface between the kernel itself and its peripheral devices. Device drivers are thus crucial elements in SunOS's overall device-independent scheme of things. Device-drivers are the *only* parts of the system that know, or care, if a device is DMA (Direct Memory Access), PIO (Programmed I/O), or memory-mapped.

The kernel supplied with the Sun system is a *configurable* kernel, meaning that it is possible to add new device driver modules to your system by rebuilding your kernel, even if you don't have access to the system source code. On Sun386i systems, the loadable driver capability makes it possible to attach a driver to a system without rebuilding the kernel and rebooting the system. For more information on how to reconfigure your kernel to include new device drivers, see the *Configuring the Kernel* and *SunOS STREAMS Topics* chapters of this manual, the *Adding Hardware to Your System* chapter of *Network Programming* and the `config(8)` man page.

1.2. Types of Devices

This document is aimed at Sun users who wish to connect new Multibus, VMEbus or ATbus devices to their system. It does *not*, however, explain how to write drivers for all possible Sun devices.

We can classify devices into eight major categories:

1. Co-processors.
2. Disks and tapes.
3. Network interface drivers such as Ethernet or X.25.
4. SCSI devices.
5. Serial communications multiplexors.
6. General DMA devices such as driver boards for raster-oriented printers or plotters. DMA devices contain their own processors and, once dispatched, perform I/O independently of the system CPU by stealing memory cycles.
7. Programmed I/O devices, that is, devices which send and receive data on the main system bus under direct control of the system CPU.
8. Frame buffers and other memory-mapped devices. Such devices are typically mapped into user-process memory and then accessed directly.
9. So called *pseudo devices*, which are actually drivers without associated hardware devices.

This manual does *not* cover driver development for devices in categories 1, 2, 3, 4 and 5. It does discuss — in Part one — drivers for the devices in categories 6, 7, 8 and 9 and — in Part Two — gives STREAMS-related information of interest to programmers planning drivers for serial communications devices. The majority of the devices which users will want to add to their systems are found in categories 6 to 9. These include:

- input devices like mice, digital tablets and analog-to-digital converters,
- output and display devices like frame buffers, printers, and plotters,
- utility peripherals like array and graphics processors.

This manual doesn't support the development of co-processor drivers for the simple reason that co-processors, while certainly devices, are so intimately linked to the CPU that they are integrated below the driver level of the kernel.

It also excludes tape and disk drivers, or indeed drivers for any *structured* or *block I/O* devices, for such drivers are quite difficult to write well. Besides, most customers will find that the structured-device drivers provided with the standard system software fill their needs quite adequately. The extensive use of standards within the Sun product line will allow them to use hardware interfaces already provided by Sun to drive whatever tape and disk units they wish to use. If this turns out not to be the case, an experienced driver developer will have to be consulted. (You will also want to start with an existing driver, and will thus need a source-code license).

Finally, this manual doesn't really discuss the issues relevant to serial communications and local network interface driver development. Again, such drivers are rather involved, and users will almost certainly find the Sun product line to contain devices adequate to their task. (And again, you will need a source license to go it alone).

This manual is primarily concerned with *unstructured* or *character* (as opposed to *structured* or *block*) devices. This distinction is often made, but seldom clearly, and it may be helpful then to consider *structured* devices as only those upon which SunOS filesystems can be mounted. Such devices (almost always disks, but tape drives are possible) support random-access I/O by way of the system buffer-caching mechanism. They almost always support a second, character-oriented style of I/O, often called *raw I/O*, but this doesn't make them character devices. Their drivers tend to implement raw I/O with the same mechanisms constructed for the main task of supporting block I/O.

Character devices, on the other hand, do not support random-access I/O, and filesystems cannot be mounted upon them. Their drivers typically support *read* and/or *write* operations, but these operations are fundamentally different than in block devices. *Sometimes character drivers use mechanisms, routines and structures that are primarily intended for block drivers, but this shouldn't be allowed to confuse matters; they use them only because it's convenient to do so.*[1]

The techniques described in this manual can also be used to build *pseudo-device drivers*. Such drivers can be useful in a variety of ways. They can be used to implement virtual devices (for example, windows that behave as virtual terminals) or for extending the capabilities of the kernel in highly localized and portable fashions (for example, by building a pseudo device to implement a specific kind of semaphore facility). What they all have in common is the absence of hardware; the driver actually implements and controls virtual software devices.

---

[1] To jump ahead for a moment, the kernel routines which, though written for block drivers are also used for character drivers are `physio()`, `mbsetup()` and `mbrelse()`. The driver *xx*`strategy()` routine is also intended primarily for block devices, though it can be used in character drivers which buffer their I/O (typically those which don't support a tty-style interface). In such cases it's not, as it is in block drivers, an entry point, and it doesn't implement any strategy to speak of. But `physio()` requires its existence, as it does the use of the `buf` structure, and so they are used. The main point to keep in mind is that character drivers use block-driver mechanisms because it's convenient for them to do so, but this doesn't make them block drivers. In particular, character drivers never have anything to do with the kernel buffer cache.

1.3. System V Compatibility

The SunOS applications interface is almost completely compatible with that of AT&T's System V UNIX system. The driver/kernel interface, however, is not. In general, though, drivers that were written for System V (or V7 or 4.1BSD, which have driver interfaces similar to System V) will be easily ported to SunOS, because, with the exception of drivers for pseudo devices, drivers are far more sensitive to the architectural details of the machines upon which they run than to the details of the kernels to which they interface.

Sun device drivers differ from typical System V drivers because the Sun operating system has evolved from 4.2BSD and, in 4.2BSD, the kernel driver interface was significantly restructured. This doesn't mean that programmers with experience developing System V drivers will find Sun drivers to be altogether foreign. In fact, the overall structure of Sun drivers is largely identical to the structure of System V drivers. Nevertheless, there are differences, and from some perspectives they are quite significant. See the *Overall Kernel Context* chapter of this manual for the details of the Sun driver/kernel interface.

The greatest differences between Sun drivers and drivers for other systems are due not to operating system differences but rather to differences between the Sun Memory-Mangement Unit (MMU) and the MMUs of other systems. Consequently, drivers which map addresses require a lot of Sun-specific code.

1.4. Major Development Stages

To add a new device and its driver to the system you must:

1. Get the device hardware into a state where you know it works as advertised. It is *extremely* difficult to debug the driver software if the device hardware isn't first working properly.
2. Write the device driver itself.
3. Add the driver to a kernel's configuration file to specify a system containing the new driver, and compile this system. On the Sun386i, if you have written the driver as a loadable driver, then compile the driver and use the `modload(1)` command to load the driver into a running system.
4. Debug the driver.
5. Repeat steps 2 to 4 as necessary. Drivers are often written (and debugged) by stages, with development proceeding long after early versions are configured into the kernel.

1.5. Warning To Microcomputer Programmers

Sun computers are virtual-address machines, and, as such, their addressing schemes are far more complex than anything that microcomputer programmers typically confront. In virtual-address machines, physical addresses have a complex and rapidly changing relationship to the virtual addresses which user programs manipulate. The kernel continually maps, remaps and unmaps pages of virtual memory to accommodate the limits of system physical memory. This means that the kernel (including its device drivers) cannot assume that any physical address in user memory will not be snatched away by the paging daemon unless it explicitly locks the physical page containing that address into memory. The details of how this locking is done will be given later, in discussions of the kernel support routine `physio()`; for the moment simply note that physical addresses have a complex and transient relationship to virtual addresses. Specifically:

- Each user process (and, on Sun-2 machines, the kernel as well) has its own distinct virtual address space. A user process (or the kernel) can make arrangements to share address space with another process — that is, to have part of its address space mapped to the same physical memory as a part of the address space of another process — but this must be done explicitly.
- In similar regard, a user process can elect to have a bus address mapped into its address space, but this doesn't happen automatically.

1.6. Address-Space Terminology

In this manual, we will adopt a VMEbus address-space naming convention that makes both address size and data size explicit. The first number in the name indicates the number of bits in the address and the second number indicates the number of bits in the data length. For example, the space with a 24-bit address and a 16-bit data length will be known as `vme24d16`. This naming convention is used elsewhere, but others are as well, as indicated in the following table.

Table 1-1 *VMEbus Address-space Names*

Address-Space Name	Other Name(s)
`vme16d16`	VME D16A16 and `vme16`
`vme24d16`	VME D16A24 and `vme24`
`vme32d16`	VME D16A32
`vme16d32`	VME D32A16
`vme24d32`	VME D32A24
`vme32d32`	VME D32A32 and `vme32`

The short names in the second column (`vme16`, `vme24` and `vme32`) are commonly used, but they can seem ambiguous to the novice, and will consequently be avoided in this manual.

Note that there are two situations where the system expects the name of a VMEbus address space as input. In these situations, either the `vme16d16` or the `vme16` forms are acceptable. These situations are:

- within the kernel config file, and
- when naming actual memory devices ("special" files in the `/dev` directory). See the *Mapping Devices Without Device Drivers* section of the *Driver*

Development Topics chapter for more information.

1.7. Manual Overview

Regular Drivers

Chapter 2 is an overview of the hardware environment provided by Sun Workstations to their drivers. The emphasis is on bus and address-space related issues.

Chapter 3 is an overview of the kernel environment within which drivers operate.

Chapter 4 covers a number of topics relevant to drivers: address spaces, interrupts and so on, in greater detail. It also surveys the most important classes of services provided by the kernel to its drivers.

Chapter 5 covers development topics, including the initial installation and checkout of devices, driver debugging and error handling.

Chapter 6 provides a detailed discussion of a driver for a very simple hypothetical character device.

Chapter 7 explains how to add new drivers to the SunOS kernel.

Chapter 8 explains pseudo-drivers, and provides source and installation instructions for a real ramdisk pseudo-driver.

STREAMS Drivers

Chapter 9 is and introduction to the STREAMS mechanism.

Chapter 10 describes the development of user-level STREAMS applications.

Chapter 11 discusses, in detail, the development of STREAMS drivers and modules.

Chapter 10 discusses those aspects of the STREAMS mechanism that are unique to SunOS. It covers the few STREAMS–specific configuration topics.

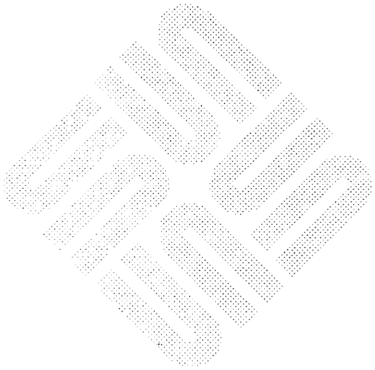
Finally, there are appendices containing information useful to driver developers. These include a set of STREAMS-specific appendices (included in Part II), a summary of kernel support functions useful in developing device drivers, descriptions of user-level routines useful in driver development, and a number of annotated driver listings.

Last Word

Remember, spend as much time as you need in the Sun PROM monitor poking, prodding and cajoling your device until you're thoroughly familiar with its behavior. This will save you a lot of grief later. The details on how to proceed with a monitor checkout of your device are found in the *Installing and Checking the Device* section of the *Driver Development Topics* chapter.

And finally, note that if you have no previous experience writing UNIX device drivers, you should expect to seek some help from the Sun technical support or consulting organizations, or from an outside consultant experienced with driver development.

PART ONE: Regular Device Drivers



2

Hardware Context

Hardware Context 13

2.1. Multibus Machines 13
  Multibus Memory Address Space and I/O Address Space 13
  Allocation of Multibus Memory 16
  Allocation of Multibus I/O Space 17
2.2. VMEbus Machines 18
  Sun-2 VMEbus Address Spaces 18
  Sun-3/Sun-4 Address Spaces 20
  Allocation of VMEbus Memory 22
  The Sun VMEbus to Multibus Adapter 24
  Interrupt Vector Assignments 24
2.3. ATbus Machines 25
  Loadable Drivers 27
  DOS and SunOS Environments 27
2.4. Hardware Peculiarities to Watch Out For 28
  Multibus Device Peculiarities 28
    Multibus Byte-Ordering Issues 28
    Other Multibus-related Peculiarities 30
  Sun-4/SPARC Peculiarities 31
  Other Device Peculiarities 32
2.5. DMA Devices 33
  Sun Main-Bus DVMA 33
  DMA on ATbus Machines 36

2

Hardware Context

Computer I/O architectures are far more dependent upon bus structure than they are upon CPU type, and device drivers, oriented as they are towards I/O, must have intimate knowledge of the bus characteristics of the machines on which they are running. For example, many Multibus machines do not support vectored interrupts [2] and thus drivers for interrupt driven devices which are intended to run on Multibus machines must provide polling facilities. Fortunately, the Sun kernel provides facilities (described in the *Other Kernel/Driver Interfaces* section of the *Overall Kernel Context* chapter) by which a driver can determine the type of the machine upon which it's running.

2.1. Multibus Machines

Multibus Memory Address Space and I/O Address Space

The MC680X0 family of processors does all its I/O via a process known as "memory mapping." What this means is that the processor sees no difference between memory and peripheral devices — all input-output operations are performed by storing data and fetching data from the same memory space. The Multibus, on the other hand, was originally designed for processors, like those of the Intel 8080 family, which have two separate address spaces. Such processors have one kind of instruction for storing data in memory or fetching data from memory (instructions such as MOV), and another, different kind of instruction (such as IN and OUT) for transferring data to or from peripheral devices. Reflecting the architecture of such processors, the Multibus has two address spaces.

Multibus memory space

> is used for memory or devices that look like memory. Many devices — commonly known as "memory mapped" devices — are designed to be accessed as memory, and drivers for such devices can "map" them into user virtual memory space and then perform device I/O by simply reading and writing the device's memory as part of normal address space. Such memory-mapped drivers tend to be quite simple, and so it's notable that devices not explicitly designed to be memory mapped can, under a restricted set of circumstances, be driven by memory mapping. The restrictions are,

---

[2] The Multibus itself, as it turns out, actually does support vectored interrupts, but not in a way that can reasonably be used with the MC680X0 family of processors.

however, fairly severe. Such drivers cannot, for example, have `xxioctl()` routines. See the *Mapping Devices Without Device Drivers* section of the *Driver Development Topics* manual for more details. The Sun-2 Color Board is a good example of a device that *is* designed to be memory mapped, and a listing of its driver can be found in the *Sample Driver Listings* appendix.

Multibus I/O address space
is another "space" entirely separate from normal memory. Typically used as an area to which device registers can be mapped, I/O space was originally introduced to keep such registers out of limited primary address space by providing a means of making peripherals, rather than system memory, respond to the bus whenever given I/O control lines were asserted by the CPU. (Such a setup also reduces hardware costs by keeping the number of address lines small.) Devices which have their control and status registers mapped to Multibus I/O address space are said to be "I/O mapped" devices.

The MC680X0 family, of course, no longer suffers the addressing limitations that made the dual-space architecture of the Multibus so attractive. The VMEbus, in similar regard, is no longer structured around separate "memory" and "I/O" spaces. (The term "I/O space" does continue to be used, from time to time, with reference to VMEbus-based systems and devices. Such use, however, is largely by way of analogy with Multibus systems, and it shouldn't be taken too literally).

Be aware that generic Multibus memory space can be either 20-bit or a 24-bit. (Sun normally uses 20-bit Multibus memory addresses, though when a Multibus card is installed in a VMEbus system with a VMEbus/Multibus adapter, 24-bit addresses are used). In similar regard, a generic Multibus can provide either an 8-bit or 16-bit I/O space, and Sun uses only the 16-bit Multibus I/O space. Note, however, that some older Multibus boards accept only 8-bit Multibus I/O addresses.

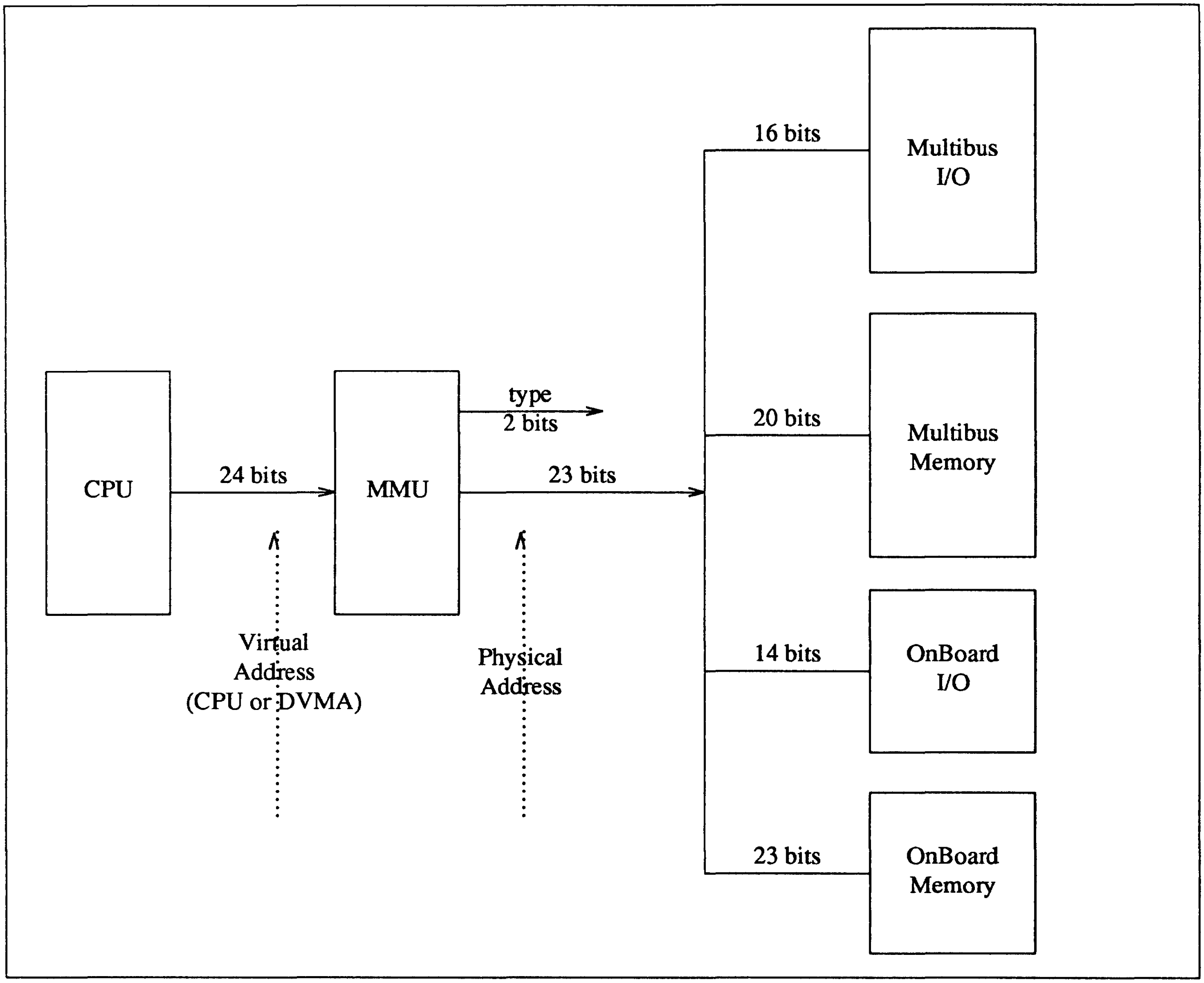
Sun Multibus systems actually have four "address spaces," corresponding to the four types of memory (each type has an identifying number associated with it, a number which is used by the MMU in computing PTE's (Page Table Entries). See the *Sun-2 Address Mapping* section of the *Driver Development Topics* chapter for details. Though you will seldom deal with the on-board address spaces, you're best off understanding what they are. The following table thus contains not only the two Multibus spaces, but the "on board" memory and I/O spaces as well. It's within these spaces, resident on the CPU board itself, that SunOS is run.

Table 2-1 *Sun-2 Multibus Memory Types*

Type	*Description*	*Address Size*	*Address Range*
0	On-Board Memory	23 bits	`0x0 – 0x7FFFFF`
1	On-Board I/O Space	14 bits	`0x0 – 0x3FFF`
2	Multibus Memory	20 bits	`0x0 – 0xFFFFF`
3	Multibus I/O Space	16 bits	`0x0 – 0xFFFF`

The following schematic view of the Sun-2 Multibus may help the driver developer to visualize the larger hardware context within which drivers operate (when running on a Sun-2 Multibus machine.)

Figure 2-1 *Sun-2 Multibus Address Spaces*



CPU
24 bits
MMU
type
2 bits
23 bits
Virtual Address (CPU or DVMA)
Physical Address
16 bits
Multibus I/O
20 bits
Multibus Memory
14 bits
OnBoard I/O
23 bits
OnBoard Memory

Note some significant aspects of addressing layout as indicated in this table.

- The Memory Management Unit is at the *center* of the picture, a position that reflects its importance in the addressing scheme of all Sun machines, VMEbus based as well as Multibus based. (The centrality of the MMU will become quite clear when you later set out to allocate a physical address to your device, and then examine/set it with the PROM monitor.)

- Secondly, the input address of the MMU is a 24-bit *virtual address*. It may originate with the CPU, or come from a DMA bus master; it makes no difference.
- The output is a 23-bit *physical address* and a 2-bit *address type*. The address type specifies one of the four address spaces indicated at the right of the diagram.
- The four address spaces are to the right. The space corresponding to the incoming virtual address is a function of both the address and the memory type. Note that only the top two memory spaces (Multibus I/O and Multibus Memory) are accessible by way of the Multibus; the two On-Board memory spaces are accessed directly and are seldom of concern to non-Sun driver developers.

Programs can only reference driver address spaces in terms of virtual addresses which are then translated by the MMU into physical addresses within the appropriate physical address space.

Allocation of Multibus Memory

Here are some notes about the allocation of Multibus Memory resources in the Sun system.

No devices may be assigned addresses below `0x40000` in Multibus memory space since the CPU uses these addresses for DVMA. (See the end of this chapter for a discussion of DVMA).

The table on the next page shows a map of how Multibus Memory space is laid out in the Sun system. Note that this memory map, as well as all of those that follow, is only a general guide. To be sure that you are not installing a device at a location that will put it in conflict with existing devices, it's necessary to check the configuration of the specific systems into which it will be installed. The best way to do so is to check the local config file for the physical addresses of the devices installed within the bus of interest. This will probably give you enough information, but if you still think that there may be a conflict, and if you have a Sun source license, you can check the driver header files to determine the amount of space consumed on the bus by existing devices. With the exception of the Sky board, these devices can be rearranged. Also note the possibility that your machine will have devices attached to it, and taking up bus space, even though those devices do not appear in the config file. This possibility exists because the *xx*`mmap()` system call can sometimes be used to drive a device without installing it in the formal sense — see the *Mapping Devices Without Device Drivers* section of the *Driver Development Topics* chapter for more details.

Table 2-2 *Sun-2 Multibus Memory Map*

Address	*Device*
0x00000 – 0x3FFFF	DVMA Space (256 Kilobytes)
0x40000 – 0x7FFFF	Sun Ethernet Memory (#1) (256 Kilobytes)
0x80000 – 0x83800	SCSI (#1) (16 Kilobytes)
0x84000 – 0x87800	SCSI (#2) (16 Kilobytes)
0x88000 – 0x8B800	Sun Ethernet Control Info (#1) (16 Kilobytes)
0x8C000 – 0x8F800	Sun Ethernet Control Info (#2) (16 Kilobytes)
0x90000 – 0x9F800	*** FREE *** (64 Kilobytes)
0xA0000 – 0xAF800	Sun Ethernet Memory (#2) (64 Kilobytes)
0xB0000 – 0xBF800	*** FREE *** (64 Kilobytes)
0xC0000 – 0xDF800	Sun Model 100/150 Frame Buffer (128 Kilobytes)
0xE0000 – 0xE1800	3COM Ethernet (#1)
0xE2000 – 0xE3800	3COM Ethernet (#2)
0xE4000 – 0xE7C00	*** FREE *** (16 Kilobytes)
0xE8000 – 0xF7800	Reserved for Color Devices (64 Kilobytes)
0xF8000 – 0xFF800	*** FREE *** (16 Kilobytes)

Allocation of Multibus I/O Space

Multibus I/O address space is specified in the config file as `mbio`. From the PROM monitor, Multibus I/O space begins at `0xEB0000`, and extends to `0xEC0000`.

Prior to Sun Release 3.0, the system made the assumption that any address lower than `0x10000` that it found in its config file was a Multibus I/O address. With current releases this is no longer true; now the bus type of every address must be explicitly given.

The following table of generic Multibus I/O usage, like the table above, is intended only as a guide.

Table 2-3 *Sun-2 Multibus I/O Map*

Address	*Device Type*
0x0040 – 0x0047	Interphase Disk Controllers
0x00A0 – 0x00A3	CPC TapeMaster Controllers
0x0200 – 0x020F	Archive Tape Drives
0x0400 – 0x047F	Ikon 10071-5 Multibus/Versatec Interface
0x0480 – 0x057F	Systech VPC-2200 Versatec/Centronics Interfaces
0x0620 – 0x069F	Systech MTI-800/1600 terminal Interface
0x2000 – 0x200F	Sky Board
0xEE40 – 0xEE4F	Xylogics 450/451 Disk Controller
0xEE60 – 0xEE6F	Xylogics 472 Multibus Tape Controller

2.2. VMEbus Machines

VMEbus machine architecture is generally more complex than Multibus machine architecture — it makes no distinction between I/O space and Memory space, but on the other hand it supports multiple address spaces. It does so for reasons of both cost and flexibility. The VMEbus was designed to be cost-effective for a range of applications. It is expensive (in terms of money, power, and board space) to provide the hardware for a full 32-bit address space. If installed devices only respond to 16-bit addresses, it makes sense to be able to put them all into a 16-bit address space and save the cost of 16-bits' worth of address decoders and the like. The 24 and 32-bit address spaces are similar compromises between cost and flexibility.

The driver writer has to understand which address space his board uses (generally, this is completely out of his/her control), and make an appropriate entry in the config file. For DMA devices, the driver writer has to know the address space that the board uses for its DMA transfers (this is usually a 32 or 24-bit space).

Sun-2 VMEbus Address Spaces

The Sun-2 VMEbus machines are based upon the 24-bit subset of the generic VMEbus — they support only a 16-bit and a 24-bit address space. These address spaces are known as `vme16d16` (16 data bits and 16 address bits) and `vme24d16` (16 data bits and 24 address bits). Sun-2 VMEbus machines also contain on-board memory and I/O space, of course, but these aren't accessed by way of the VMEbus and are only barely relevant to the driver developer.

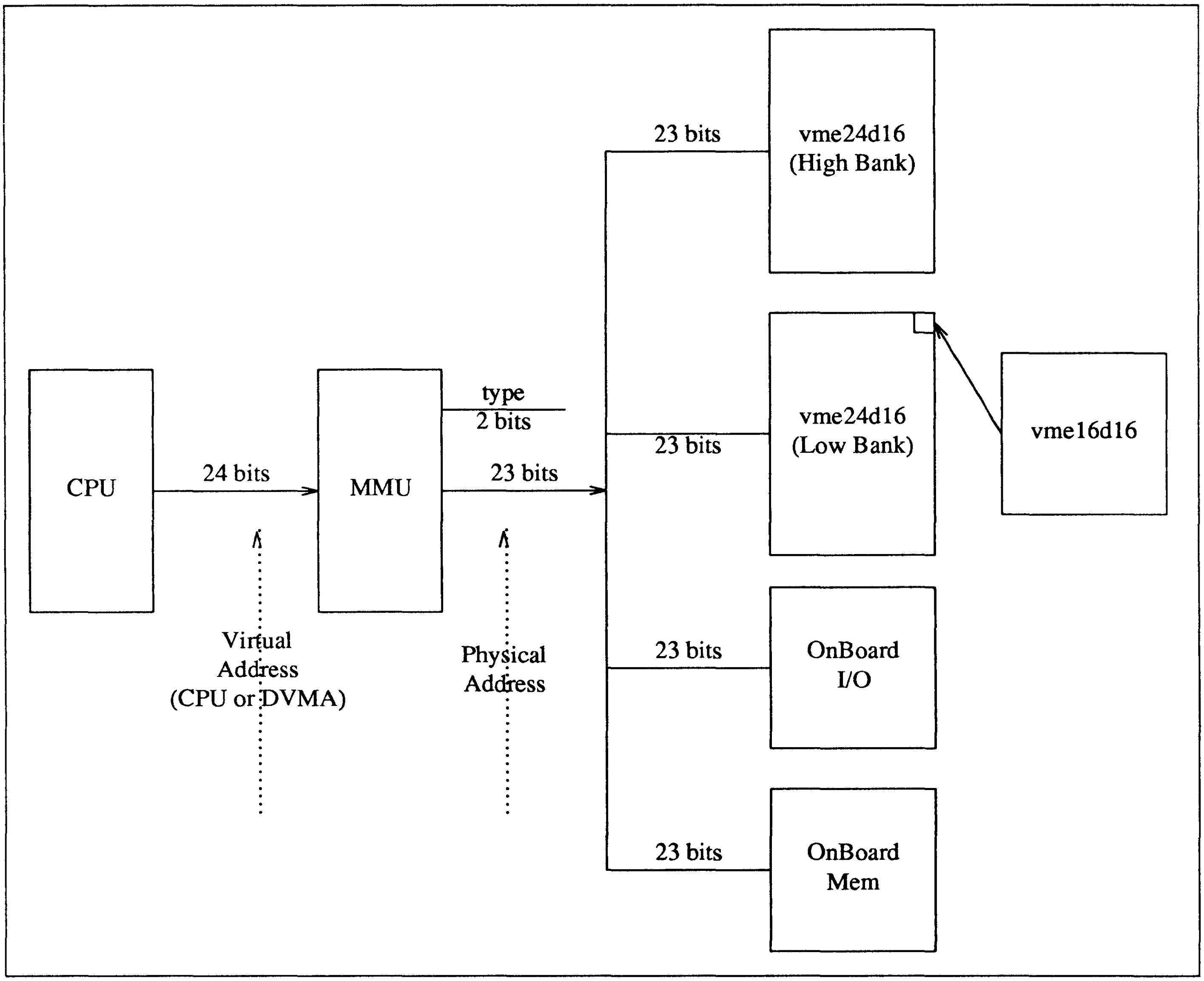
There are four types of memory on Sun-2 VMEbus machines:

Table 2-4 *Sun-2 VMEbus Memory Types*

Description	*Address Size*	*Address Range*
On-Board Memory	23 bits	0x0 - 0x7FFFFF
On-Board I/O Space	23 bits	0x0 - 0x7FFFFF
`vme24d16`	23+1 bits	0x0 - 0xFEFFFF
`vme16d16` — Stolen from top 64K of `vme24d16` (0x0 - 0xFFFF)		

The four address spaces are laid out as follows:

Figure 2-2 *Sun-2 VMEbus Address Spaces*


CPU
24 bits
MMU
type
2 bits
23 bits
Virtual
Address
(CPU or DVMA)
Physical
Address
23 bits
vme24d16
(High Bank)
23 bits
vme24d16
(Low Bank)
vme16d16
23 bits
OnBoard
I/O
23 bits
OnBoard
Mem

Note a few details:

- In all Sun-2 machines (as in Sun-3s and Sun-4s), the address input into the MMU is a virtual address, and may originate with either the CPU or a DVMA (Direct Virtual Memory Access) bus master. (See the *Sun Main-Bus DVMA* section, later in this chapter, for a discussion of DVMA).
- Unlike Sun-2 Multibus systems, in which each memory type maps cleanly to one address space, `vme24d16` maps to two different memory banks. Addresses from `0x0` to `0x7FFFFF` are "type 2" memory, while those from `0x800000` and up are "type 3". This is because Sun-2 VMEbus machines have only 23 output address bits, and this trick is necessary to generate the full range of a 24-bit address space. (See *Sun-2 Address Mapping* in the

Driver Development Topics chapter for more details).

- Multibus boards, connected to VMEbus to Multibus adapters, can be plugged into physical memory anywhere within `vme24d16` (which means that they can also be in `vme16d16`).
- The 24 bits in the `vme24d16` address space are referred to in the above table as 23+1 bits. This is because, as should be clear in the diagram above, the Sun-2 MMU outputs only the lower 23 bits of the address, and the 24th bit is actually one of the MMU's type bits.
- Note especially that `vme16d16` *is stolen from* `vme24d16`. It's selected by addresses in the form `0xFFXXXX`, that is, addresses which have the 8 high bits set.

Sun-3/Sun-4 Address Spaces

Sun-3 and Sun-4 machines are all based on the full 32-bit VMEbus, so let's begin their discussion with a listing of the address types supported by the generic VMEbus.

Table 2-5 *Generic VMEbus (Full Set)*

VMEbus-Space Name	*Address Size*	*Data Transfer Size*	*Physical Address Range*
`vme32d16`	32 bits	16 bits	`0x0 – 0xFFFFFFFF`
`vme24d16`	24 bits	16 bits	`0x0 – 0xFFFFFF`
`vme16d16`	16 bits	16 bits	`0x0 – 0xFFFF`
`vme32d32`	32 bits	32 bits	`0x0 – 0xFFFFFFFF`
`vme24d32`	24 bits	32 bits	`0x0 – 0xFFFFFF`
`vme16d32`	16 bits	32 bits	`0x0 – 0xFFFF`

Not all of these spaces are commonly used, but they are all nevertheless supported by the Sun-3 and Sun-4 lines. The following table indicates their sizes and physical address mappings.

Table 2-6 *Sun-3/Sun-4 VMEbus Address Types*

Type	*Address-Space Name*	*Address Size*	*Address Range*
0	On-board Memory	32 bits	`0x0 – 0xFFFFFFFF`
1	On-board I/O	24 bits	`0x0 – 0xFFFFFF`
2	`vme32d16`	32 bits	`0x0 – 0xFEFFFFFF`
3	`vme32d32`	32 bits	`0x0 – 0xFEFFFFFF`
2	`vme24d16` — Stolen from top 16M of `vme32d16` (0x0 - 0xFEFFFF)		
2	`vme16d16` — Stolen from top 64K of `vme24d16` (0x0 - 0xFFFF)		
3	`vme24d32` — Stolen from top 16M of `vme32d32` (0x0 - 0xFEFFFF)		
3	`vme16d32` — Stolen from top 64K of `vme24d32` (0x0 - 0xFFFF)		

Sun-3/Sun-4 space overlays are much more complex than those of the Sun-2, as is evident from both the table above and the diagram below. The principle, however, is the same — when a space overlays a larger space, its memory is stolen from that larger space and is considered by the MMU to be in the the overlaid space. One simply cannot address above `0xFF000000` in 32-bit VMEbus space or above `0xFF0000` in 24-bit VMEbus space.

As the two following diagrams illustrate, Sun-3 and Sun-4 addressing schemes are almost identical. They differ only in the size of the virtual address which — output by the CPU or a DVMA Bus Master — is fed to the MMU.

Figure 2-3 *Sun-3 VMEbus Address Spaces*

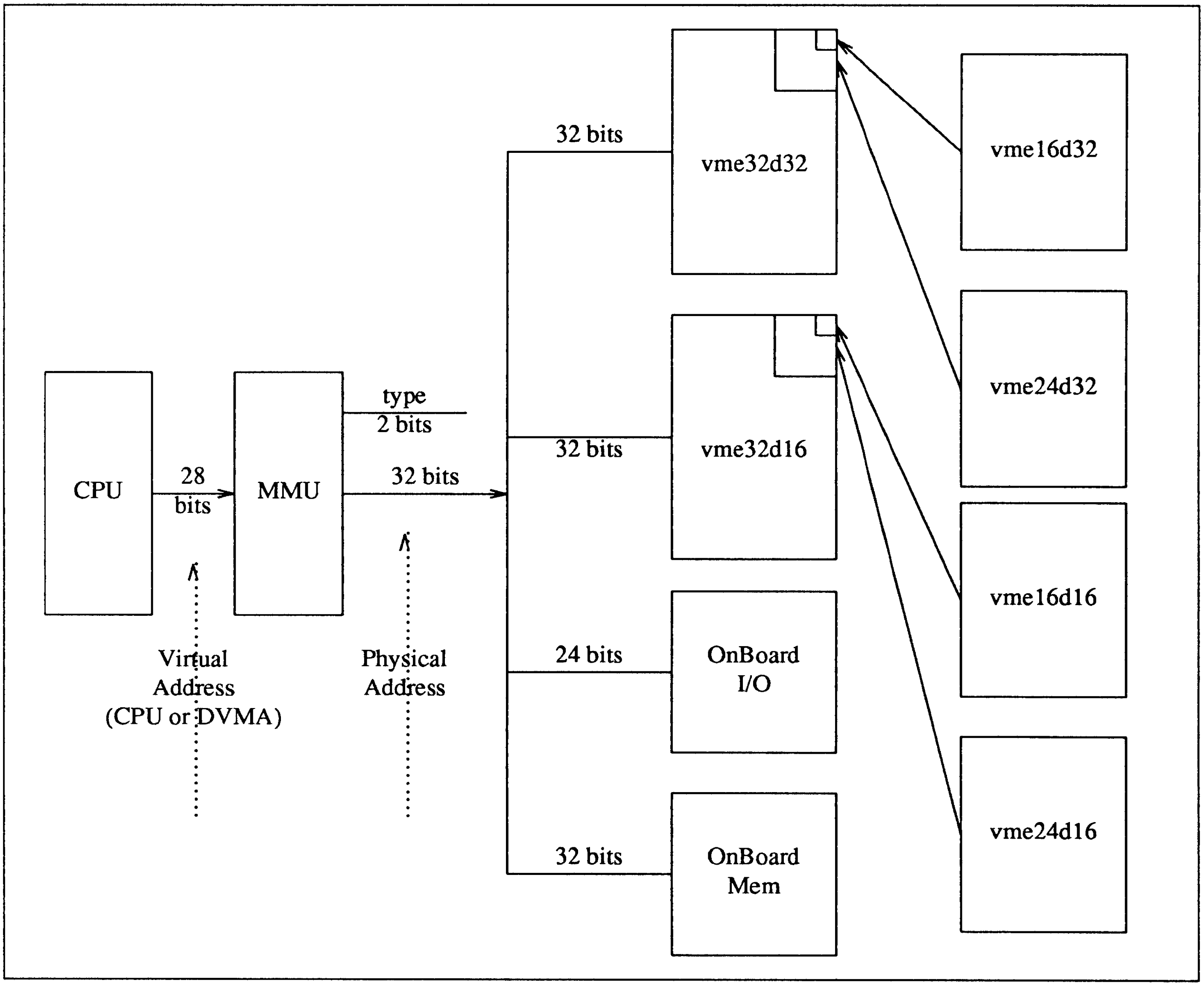
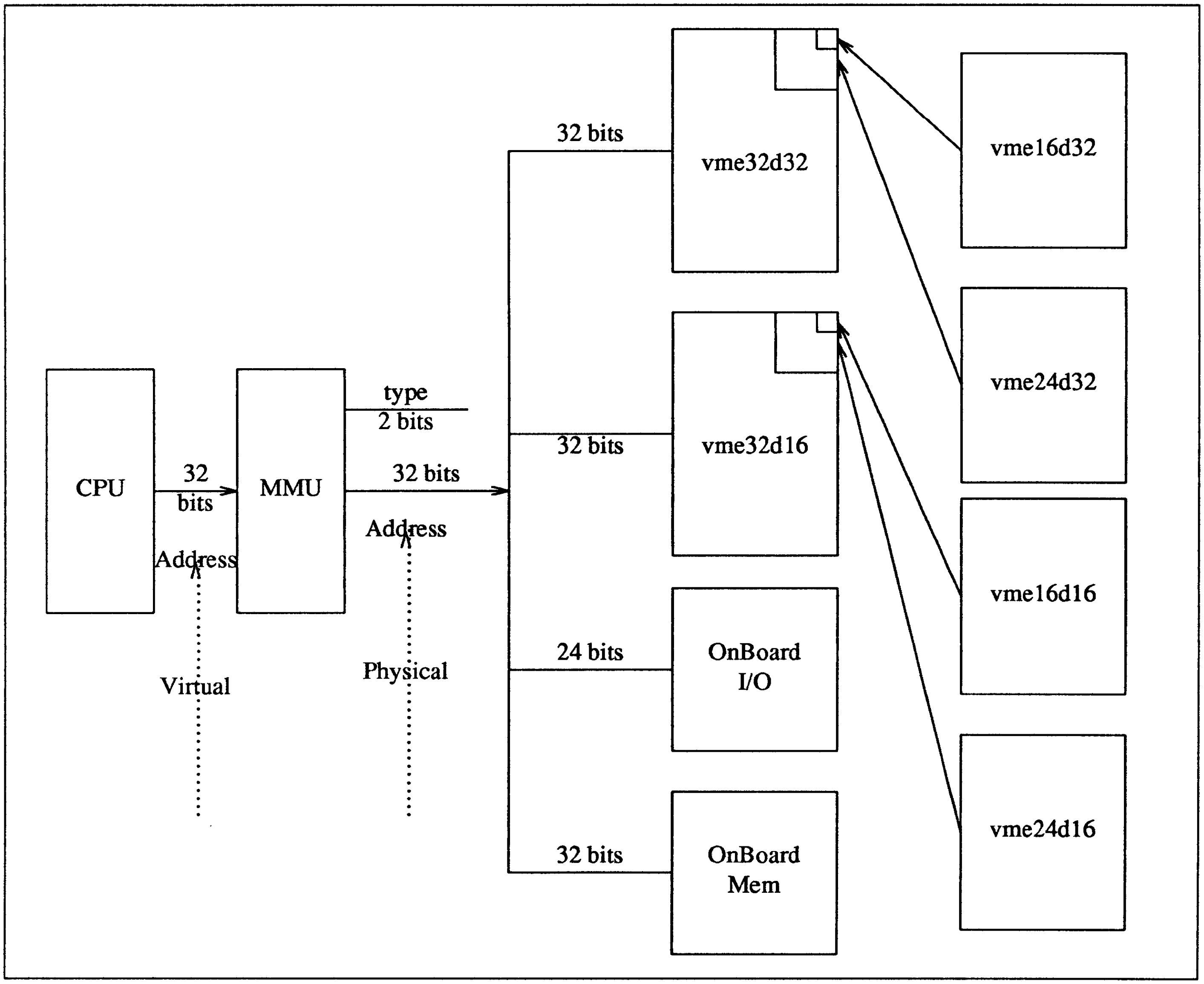

CPU
28 bits
MMU
type 2 bits
32 bits
Virtual Address (CPU or DVMA)
Physical Address
32 bits
vme32d32
32 bits
vme32d16
24 bits
OnBoard I/O
32 bits
OnBoard Mem
vme16d32
vme24d32
vme16d16
vme24d16

Figure 2-4 *Sun-4 VMEbus Address Spaces*


CPU
32 bits
Address
Virtual
MMU
type
2 bits
32 bits
Address
Physical
32 bits
32 bits
24 bits
32 bits
vme32d32
vme32d16
OnBoard I/O
OnBoard Mem
vme16d32
vme24d32
vme16d16
vme24d16

Allocation of VMEbus Memory

This section summarizes the typical use of the 16, 24 and 32-bit VMEbus address spaces by Sun devices. Note well that the usages summarized here are only for the generic configuration, and there's no guarantee that they match the exact usage on your machine. They will, however, help you to decide where to attach your device. The "Allocated From" field shows whether bus space is allocated from the high end of the given range or from the low end. The idea is to keep the maximum size "hole" in the middle in case the boundary needs to be shifted later.

Table 2-7 *16-bit VMEbus Address Space Allocation*

Address Range	*Allocated From*	*Description of Use*
`0x0000-0x7FFF`	Low	Reserved for OEM/user devices
`0x8000-0xFFFF`	High	Reserved for Sun devices

16-bit VMEbus space is mapped into the topmost 64K of 24-bit VMEbus space at `0x00FF0000` to `0x00FFFFFF` (on Sun-2s) or `0xFFFF0000` to `0xFFFFFFFF` (on Sun-3s and Sun-4s). Note: The Multibus/VMEbus Adapter will map the Multibus I/O addresses of Multibus cards that use Multibus I/O into the same addresses in the 16-bit VMEbus space. This may place the standard Multibus addresses for some cards into the OEM/user area in the above table. These addresses can be changed, if necessary, by physically readdressing the device and then changing its entry in the config file.

Table 2-8 *24-bit VMEbus Address Space Allocation*

Address Range	*Allocated From*	*Description of Use*
`0x000000-0x0FFFFF`		CPU board DVMA space
`0x100000-0x1FFFFF`		Reserved by Sun
`0x200000-0x2FFFFF`	Low	Reserved for small Sun devices
`0x300000-0x3FFFFF`	High	Reserved for large Sun devices
`0x400000-0x7FFFFF`	(Taken)	Reserved for huge Sun devices
`0x800000-0xBFFFFF`	High	Reserved for huge OEM/user devices
`0xC00000-0xCFFFFF`	Low	Reserved for large OEM/user devices
`0xD00000-0xDFFFFF`	High	Reserved for small OEM/user devices
`0xE00000-0xEFFFFF`		Multibus-to-VMEbus memory space
`0xF00000-0xFEFFFF`		Reserved for the Future
`0xFF0000-0xFFFFFF`		Stolen by 16-bit VMEbus space

Table 2-9 *32-bit VMEbus Address Space Allocation (Sun-3s and Sun-4s Only)*

Address Range	*Description of Use*
`0x00000000 - 0x000FFFFF`	DVMA Space
`0x00100000 - 0x7FFFFFFF`	Reserved by Sun
`0x80000000 - 0xFEFFFFFF`	Reserved for OEM/user devices
`0xFF000000 - 0xFFFFFFFF`	Stolen by `vme24d16`

These same assignments apply to both 16-bit-data and 32-bit-data VMEbus accesses. Note that, at least in the GENERIC kernel, there are some Sun devices (`tm0, tm1, vpc0, vpc1 and mti0-4`) installed in the OEM/user area. It's always best to check, when choosing an installation address, that you aren't going to conflict with an already installed device.

Table 2-10 *VMEbus Address Assignments for Some Devices*

Device	*Addressing*	*Addresses Used*
VMEbus SKY Board	`vme16d16`	0x8000 - 0x8FFF (Sun-2 only)
VMEbus SCSI Board	`vme24d16`	0x200000 - 0x2007FF
VMEbus TOD Chip	`vme24d16`	0x200800 - 0x2008FF (Sun-2 only)
Graphics Processor	`vme24d16`	0x210000 - 0x210FFF
Sun-2 Color Board	`vme24d16`	0x400000 - 0x4FF7FF

The VMEbus Sky board occupies addresses `8000-8FFF` in 16-bit address space, and it requires that the high nibble of the address be '8'. Unlike other pre-installed devices, it cannot be moved.

This table is, of course, not complete. There is always a variety of devices on the bus, as can be easily determined by examining the config file. This table, however, does include the standard devices that use a significant amount of space on the VMEbus. Note that, in machines which came after the Sun-2 line, several of these devices have been replaced by on-board devices and have thus disappeared from the VMEbus address space.

The Sun VMEbus to Multibus Adapter

Multibus devices that are to be attached to VMEbus machines must be attached to a VMEbus to Multibus adapter. (The Adapter works for most, but not all, Multibus boards). An adapter can be used to take over *one and only one* chunk of `vme24d16`. However, that chunk can overlap all or part of `vme16d16` (because `vme16d16` is a proper subset of `vme24d16`). In any case, the adapter must be told how much space the board attached to it actually expects, for by default it will take over a full megabyte. Note that the Multibus Adapter supports fully vectored interrupts, and that drivers for Multibus devices attached by way of adapters need not poll, since the adapters contain switches by which Multibus devices can be assigned vectors.

Interrupt Vector Assignments

The table below shows the assignments of interrupt vectors for those devices that can supply interrupts through the VMEbus vectored interrupt interface. To pick one for your device, examine the kernel config file for an unused number in the range reserved for customer use, `0xC8` to `0xFF`.

Table 2-11 *Vectored Interrupt Assignments*

Vector Numbers	*Description*
0x40 *thru* 0x43	sc0, sc? si0, si? — SCSI Host Adapters
0x48 *thru* 0x4B	xyc0, xyc1, xyc? — Xylogics Disk Controllers
0x4C *thru* 0x5F	future disk controllers
0x60 *thru* 0x63	tm0, tm1, tm? — TapeMaster Tape Controllers
0x64 *thru* 0x67	xtc0, xtc1, xtc? — Xylogics Tape Controllers
0x68 *thru* 0c6F	future tape controllers
0x70 *thru* 0x73	ec? — 3COM Ethernet Controller
0x74 *thru* 0x77	ie0, ie1, ie? — Sun Ethernet Controller
0x78 *thru* 0x7F	future ethernet devices
0x80 *thru* 0x83	vpc? — Systech VPC-2200
0x84 *thru* 0x87	vp? — Ikon Versatec Parallel Interface
0x88 *thru* 0x8B	mti0, mti? — Systech Serial Multiplexors
0x8C *thru* 0x8F	dcp1, dcp? — SunLink Comm. Processor
0x90 *thru* 0x9F	zs0, zs1 — Sun-3 Terminal/Modem Controller
0xA0 *thru* 0xA3	future serial devices
0xA4 *thru* 0xA7	pc0, pc1, pc2, pc3 — SunIPC
0xA8 *thru* 0xAB	future frame buffer devices
0xAC *thru* 0xAF	future graphics processors
0xB0 *thru* 0xB3	sky0, ? — SKY Floating Point Board
0xB4 *thru* 0xB7	SunLink Channel Attach
0xB8 *thru* 0xC7	Reserved for Sun Use
0xC8 *thru* 0xFF	Reserved for Customer Use

2.3. ATbus Machines

The Intel 80386 processor handles I/O devices placed in either memory space or in I/O space. On the 80386, memory-mapped I/O provides additional programming flexibility. Any memory instruction can access any I/O port located in the memory space. For example, the MOV instruction transfers data between any register and any port. The AND, OR, and TEST instructions manipulate bits in the internal registers of a device.

On some devices, reading a register will not read back what was written. Therefore, instructions such as AND, OR, and TEST can, in some cases, produce unexpected results because the instruction reads a good location, changes it, and writes it back. See the *Other Device Peculiarities* section, ahead.

Memory-mapped I/O can use the full complement of instructions. The 16 MB memory of AT memory exists in the 4 GB physical address space of the Sun386i at `0xE000 0000`. For example, a device that, on an AT, shows up in memory at `D0 0000` will show up in the Sun386i physical memory at `0xE0D0 0000`. Virtual addresses are assigned during the autoconfiguration process.

If an I/O device is mapped into the I/O space then the IN, OUT, INS, and OUTS instructions are used to communicate to and from the device. All I/O transfers are performed via the AL (8-bit), AX (16-bit), or EAX (32-bit) registers. The first 256 bytes of the I/O space are directly addressable. The entire 64 Kbyte I/O space is indirectly addressable through the DX register.

The Sun386i has 21 interrupt channels, but only 11 are available to devices on the AT bus. The following list of interrupt channel assignments shows all of the interrupt channels.

Table 2-12 *Interrupt Channel Assignments*

*AT Channel**	*Assignee*
3	AT Pin B25
4	AT Pin B24
5	AT Pin B23
6	Not available (system diskette)
7	Not available (parallel port)
8	SCSI
9	AT Pin B04
10	AT Pin D03
11	AT Pin D04
12	AT Pin D05
13	Not available (Ethernet)
14	AT Pin D07
15	AT Pin D06

* *Available to AT Cards*

When you add an AT card to the AT bus, you must select one of the values in the Channel column for the AT card's jumpers. For example, if you select channel 10 for a serial card, the "device" line in the config file might look as follows:

```
device  ns0 at atio ? csr 0x3f8 irq 10 priority 6
```

The Sun386i does not permit two AT cards to use the same interrupt channel.

Some cards will also use DMA and will have jumpers to select a DMA channel to use. The following list shows that DMA channels 0-3 and channel 5 are available for AT cards. Note that channel 0 and 5 can be used with 16-bit DMA devices; 1, 2, and 3 can be used only with 8-bit DMA devices. Note also that channels 4, 6, and 7 are pre-assigned.

Table 2-13 *Sun386i DMA Channel Assignments*

Channel	*Assignee*	*Size (bits)*
0	AT Bus	16
1	AT Bus	8
2	AT Bus	8
3	AT Bus	8
4	Software	Not Available
5	AT Bus	16
6	Ethernet	16
7	SCSI	16

For example, you might set up a controller that uses DMA channel 3. For this, the "controller" line in the config file might look like: this:

```
controller wds0 at atio ? csr 0x320 dmachan 3 irq 3 priority 3
```

The Sun386i does not permit two AT cards to use the same DMA channel.

In these examples, ''priority'' refers to the `spl` levels used in the driver. That is, the phrase ''priority 3'' implies that the driver uses `spl3()` to protect its critical regions.

Loadable Drivers

On Sun386i machines, device drivers can be dynamically loadable. That is, they can be attached to a system without rebuilding its kernel and without having to bring the system down and restart it. See the *Adding and Removing Loadable Drivers* section of the *Configuring the Kernel* chapter for details.

DOS and SunOS Environments

The Sun386i system supports both DOS drivers and SunOS drivers.

You can attach a DOS device driver in the standard DOS way, but it will be usable only from within the DOS environment. Usually, all you need to do is to first plug in an add-in board. Then you insert an installation diskette (which comes with the board) into Drive A> and re-boot the system. The device driver is already compiled and linked. Generally, the diskette contains programs called "INSTALL" or something similar. You execute this program by typing its name. It copies the driver file from the diskette to the hard disk. At the same time, this procedure will modify the disk's `config.sys` file.

The DOS system must be re-booted. The device driver will automatically be loaded into memory, its options will be parsed, and the driver will be initialized.

NOTE *The DOS driver on the Sun386i is running under SunOS and DOS, but the driver is unaware of this. SunOS might switch control to another task during device operation, so strict timing dependencies could fail. Real time devices, for example, may not work properly. If a peripheral and controller have strict timing requirements, their drivers should be written in the standard SunOS style. DOS drivers do not run at the elevated priority of SunOS drivers.*

SunOS drivers, of course, are parts of the system kernel. Thus the timing requirements of most devices can be met under SunOS. SunOS drivers are accessible from the DOS environment.

2.4. Hardware Peculiarities to Watch Out For

There is a variety of device peculiarities that the driver developer must be aware of. The most common of them are related to the Multibus and Multibus-based devices, but there are others as well.

Multibus Device Peculiarities

The IEEE Multibus is a source of problems for two separate reasons. The first of these, discussed immediately below, is the fact that the Multibus has a different notion of byte order than does the either Motorola MC680X0 family or the Sun SPARC processor (the reduced instruction set CPU upon which Sun-4 machines are built). The second is simply that the Multibus has been around for a long time, and thus brings with it a variety of older devices, many of which have addressing limitations and other characteristics which make for a less than perfect fit with the Sun architecture.

Multibus Byte-Ordering Issues

Sun-2 and Sun-3 processors are members of the Motorola MC680X0 family, while Sun-4 processors are based on the SPARC CPU. All of these processors address bytes within words by what we shall call *IBM conventions* — the most significant byte of a word is stored at the lowest addressed byte of the word. The Multibus, on the other hand, uses *DEC conventions* — the least significant byte of a word is stored at the lowest address, and significance increases with address.

This class of byte-addressing conventions leads to two separate problems, with two separate solutions:

- The first problem occurs when you're moving a single *byte* across the interface between the MC680X0/SPARC and the IEEE Multibus. Because the two devices don't agree about the end of the word that the byte actually appears in, you have to change the byte address before the move — what you want to do, in effect, is move every byte to the other side of the word which it occupies — the most CPU-efficient way of doing so is to toggle the least significant bit of every byte address.
- The second problem, also related to the Multibus, is a higher level version of the first. It occurs when machine *words* with significant internal structure (or structures that contain words) are moved across the bus interface. (If you write only words, and the device uses only words, there's no problem). The Multibus byte-ordering incompatibility will cause structures to be scrambled when they're moved across the bus interface, unless the bytes within them are physically swapped first.

Here are a few pictures describing the problems in detail:

Motorola (IBM) Byte Ordering

bit 15	bit 0
Byte 0	Byte 1

Multibus (DEC) Byte Ordering

bit 15	bit 0
Byte 1	Byte 0

That is, the MC680X0 and SPARC CPUs place byte 0 in bits 8 through 15 of the 16-bit word, whereas the Multibus places byte 1 in those bits. If you did everything with the CPU, or everything on the Multibus, there wouldn't be any conflict, since things would be consistent. However, as soon as you cross the boundary between them, the byte order is reversed. Thus, you have to toggle the least significant bit of the address of any *byte* destined for the Multibus — this will have the effect of swapping adjacent addresses and thus reordering the bytes.

To clarify this, consider an interface for a hypothetical Multibus board containing only two 8-bit I/O registers, namely a control and status register (csr) and a data register (we actually use this design later on in our example of a simple device driver). In this board, we place the command and status register at Multibus byte location 600, and the data register at Multibus byte location 601. The Multibus picture of that device looks like this:

Hypothetical Board Registers

bit 15	bit 0
Location 601 DATA	Location 600 CSR

But the MC680X0 and SPARC processors view that device as looking like this:

Hypothetical Board Registers

bit 15	bit 0
Location 600 CSR	Location 601 DATA

so that if you were to read location 600 from the point of view of the processor, you'd really end up reading the DATA register off the Multibus instead. So, when we define the *skdevice* data structure for that board, we define it by starting with the register definition in the device manual, and then swapping bytes to take account of the expected byte swapping:

```
struct skdevice {
    char    sk_data;      /* 01: Data Register */
    char    sk_csr;       /* 00: command(w) and status(r) */
};
```

This rule (flipping the least significant bit of the address) holds good for all *byte* transfers which cross the line between the MC680X0/SPARC CPU and the Multibus.

Other Multibus-related Peculiarities

- Many Multibus device controllers are geared for the 8-bit 8080 and Z80 style chips and don't understand 16-bit data transfers. Because of this, such controllers are quite happy to place what's really a word quantity (such as a 16-bit address which must be two-byte aligned in the MC680X0) starting on an odd byte boundary. Some devices use 16-bit or 20-bit addresses (many don't know about 24-bit addresses), and it often happens that you have to chop an address into bytes by shifting and masking, and assign the halves or thirds of the address one at a time, because the device controller wants to place word-aligned quantities on odd-byte boundaries. Note also that many Multibus boards are geared for the 8086 family with its segmented address scheme. An 8086 (20-bit) address really consists of a 4-bit segment number and a 16-bit address; you usually have to deal with the 4-bit part and the 16-bit part separately. For a good example of what we're talking about here, see the code for `vp.c` in the *Sample Driver Listings* appendix to this manual.
- Although there are a myriad of vendors offering Multibus products, remember that the Multibus is a "standard" that evolved from a bus for 8-bit systems to a bus for 16-bit systems. Read vendors' product literature *carefully* (especially the fine print) when selecting a Multibus board. The memory address space of the Multibus is *supposed to be* 20 or 24 bits wide and the I/O address space of the Multibus is *supposed to be* 16 bits wide. In practice, some older boards are limited to 16 bits of address space and 8 bits

of I/O space. In particular, watch for the following addressing peculiarities:

  - For a memory-mapped board, ensure that the board can actually handle a full twenty bits of addressing. Older Multibus boards often can only handle sixteen address lines. The Sun system assumes there is a 20-bit Multibus memory space out there. If the Multibus board you're talking to can only handle 16-bit addresses, it will ignore the upper four address lines, and this means that such a board "wraps around" every 64K, which means that on a Sun the addresses that such a board responds to would be replicated sixteen times through the one-megabyte address space on the Multibus. This may conflict with some other device.
  - Some Sun-2 Multibus systems, notably Sun-2/170s, have a backplane structure that complicates the installation of 24-bit memory-mapped devices. The internal "bus" on these systems (often called the P2 bus) is divided into multiple segments, each mapped to a portion of the backplane slots. In such systems, 24-bit memory-mapped devices must be installed in a different segment than that used by standard Sun-2 devices. See the *Sun-2/170 Configuration Guide* for more information.
  - For an I/O-mapped board (one that uses I/O registers), make sure that the board can handle 16-bit I/O addressing. Some older boards support only 8-bit I/O addressing. In our system, the address spaces of such boards would find themselves replicated every 256 bytes in the I/O address space. Trying to fit such a board into the Sun system would severely curtail the number of I/O addresses available in the system.

- Finally, watch out for boards containing PROM code that expects to find a CPU bus master with an Intel 8080, 8085, or 8086 on it. Such boards are of course useless in the Sun system.

Sun-4/SPARC Peculiarities

There are two peculiarities which are specific to machines built upon the Sun SPARC CPU (currently, just Sun-4s) which can impact device drivers. For more information about the Sun-4 machine architecture, see *Porting C, Fortran and Pascal Programs to the Sun-4*.

- The first problem is structure alignment. In MC680X0 family processors, structures are aligned on half-word boundaries, but on Sun-4s, the structure-alignment requirements are imposed by the most strictly-aligned structure components. For example, a structure containing only bytes and characters has no alignment restrictions, while a structure containing a `double` word must be constructed so as to guarantee that that this word falls on a 64-bit boundary.

  Programmers must be aware of these rules when writing drivers, for Sun-4 compilers will pad structures to enforce them, and such padding will not always be correct for structures intended to map to device registers. Also, structures must be carefully designed if drivers are to be portable across machine architectures.

- The second problem is data alignment. In MC680X0 family processors, characters are aligned on byte boundaries, while integers of all sizes are aligned on 16-bit boundaries. In Sun-4 machines, in contrast, all quantities must be aligned on their "natural" boundaries: 16-bit half words on 16-bit boundaries, 32-bit words on 32-bit boundaries and 64-bit `double` words on 64-bit boundaries.

In normal programs, details such as these are handled by the compiler. In drivers, however, more care must be taken. SPARC (unlike the MC68010) doesn't break down 32-bit transactions into successive 16-bit transactions. Thus, there are times when 32-bit entities have to be broken down by the driver if they are to get across the bus correctly. More specifically, 32-bit or 64-bit alignment is not possible in the 16-bit VMEbus spaces, and thus 32-bit and 64-bit data access does not exist. In the 32-bit VMEbus spaces, all data paths exist.

Other Device Peculiarities

There are other device peculiarities of interest to the driver developer. These peculiarities are particularly unfortunate in that they tend to require special handling of various kinds — byte swapping, bit shuffling, timing delays, etc. — whenever the driver contacts the device. Such special handling precludes the most obvious and desirable means of interfacing the driver to the device, by mapping the device registers into a C-structure declaration and then accessing them by way of references to structure fields.

- One of the most infuriating of these peculiarities is internal sequencing logic. Devices with this strange characteristic (a vestige of microcomputer systems with extremely limited address space) map multiple internal registers to the same externally addressable address. There are various kinds of internal sequencing logic:
  - The Intel 8251A and the Signetics 2651 alternate the same external register between *two* internal mode registers. Thus, if you want to put something in the first mode register of an 8251, you do so by writing to the external register. This write will, however, have the invisible side effect of setting up the sequencing logic in the chip so that the next read/write operation refers to the alternate, or second, internal register.
  - The NEC PD7201 PCC has multiple internal data registers. To write a byte into one of them, it's necessary to first load the first (register 0) with the number of the register into which the following byte of data will go — you then send that byte of data and it goes into the specified data register. The sequencing logic then automatically sets up the chip so that the next byte sent will go into data-register 0.
  - Another chip of a similar ilk is the AMD 9513 timer. This chip has a data pointer register for pointing at the data register into which a data byte will go. When you send a byte to the data register, the pointer gets incremented. The design of the chip is such that you *can't read the pointer register to find out what's in it*!
- In fact, it's often true that device registers, when read, don't contain the same bits that were last written into them. This means that bitwise operations (like `register &= ~XX_ENABLE`) that have the side effect of

generating register reads must be done in a software copy of the device register, and then written to the real device register.

- Another problem is timing. Many chips specify that they can only be accessed every so often. The Zilog Z8530 SCC, which has a "write recovery time" of 1.6 microseconds, is an example. This means that a delay has to be enforced (with DELAY) when writing out characters with an 8530. Things can get worse, however, for there are instances when it's unclear what delays are needed, and in such cases it's left to the driver developer to determine them empirically.
- And peripheral devices can contain chips that use a byte-ordering convention different from that used by the Sun system into which they're installed. The Intel 82586, for example, supports DEC byte-ordering conventions; this makes it perfectly compatible with Multibus-based, but not VMEbus-based, Sun machines. Drivers for such peripheral devices will have to swap bytes, as indicated above, and to take care that, in doing so, they don't inadvertently reorder the bits in any control fields greater than 16 bits in length.
- Finally, there are some common interrupt-related peculiarities worth noting:
  - When a controller interrupts, it does *not* necessarily mean that both it *and* one of its slave devices are ready. Some controllers are designed in this way, but others interrupt to indicate that the controller or one of its devices *but not necessarily both* is ready.
  - Not all devices power up with interrupts disabled and then start interrupting only when told to do so.
  - While there should be a way to determine that a board has actually generated an interrupt — an attention bit or something equivalent — some devices have no such thing.
  - Finally, an interrupting board should shut off its interrupts when told to do so (and also after a bus reset). Not all do.

2.5. DMA Devices

Many device controller boards are capable of what is known as Direct Memory Access or DMA. This means that the CPU can tell the device controller for such devices the address in memory where a data transfer is to take place and the length of the data transfer, and then instruct the device controller to start the transfer. The data transfer then takes place without further intervention on the part of the processor. When it's complete, the device controller interrupts to say that the transfer is done.

Sun Main-Bus DVMA

NOTE *Sun-2, Sun-3, and Sun-4 machines use Direct Virtual Memory Access (DVMA) to allow devices on the Main Bus (either a VMBbus or a Multibus) to perform DMA transfers from and to system virtual address space. In the Sun386i system, however, the Memory Management Unit (MMU) is incorporated directly on the Intel 80386 chip itself; devices need to use physical addresses. Sun386i DMA is* *discussed in the next Section.*

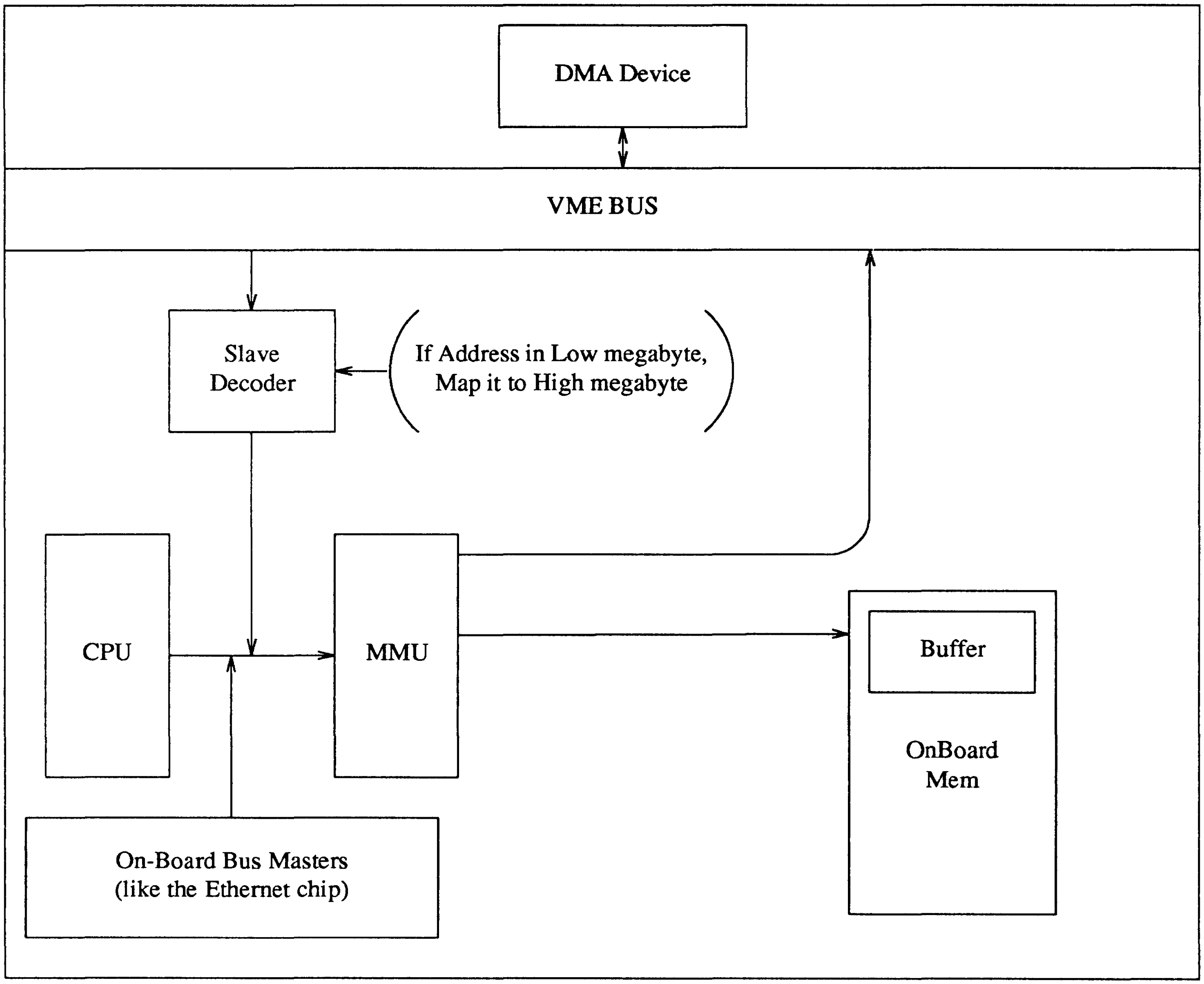
Direct Virtual Memory Access (DVMA) is a mechanism provided by the Sun Memory Management Unit to allow devices on the Main Bus (either a VMEbus or a Multibus) to perform DMA directly to Sun processor memory. It also allows Main Bus master devices to do DMA directly to Main Bus slaves without the extra step of going through processor memory. DVMA works by ensuring that the addresses used by devices are processed by the MMU, just as if they were virtual addresses generated by the CPU. This allows the system to provide the same memory protection and mapping facilities to DMA devices as it does to the system CPU (and thus to programs).

When setting up a driver to support DMA, it's necessary to know the device's DMA address size. This address size is the primary factor used in determining which of the system address spaces will host the device. Multibus devices generally have a DMA address size of 20 bits, while VMEbus devices generally have a 24 or 32-bit DMA address size.

- Since, on Sun-2 Multibus machines, DMA addresses are generally 20-bits long, the system DVMA hardware responds to the first 256K of Multibus address space (`0x0` to `0x3FFFF`). When an address in this range appears on the bus, the DVMA hardware adds `0xF00000` to it (the system places the Multibus memory address space at `0xF00000` in the system's virtual address space) and then uses the MMU to map to the location in physical memory that will be used for the data transfer.
- On Sun-2 VMEbus systems, the DVMA hardware responds to the entire lower megabyte of VMEbus address space (`0x0` to `0xFFFFF`). The system maps addresses in this range into the most significant megabyte of system virtual address space (`0xF00000` to `0xFFFFFF`).
- On both Sun-3 and Sun-4 systems, the DVMA hardware responds to the lowest megabyte of VMEbus address space *in both the 24-bit and 32-bit VMEbus spaces.* It maps addresses in this megabyte into the most significant megabyte of system virtual address space (`0xFF00000` to `0xFFFFFFF` for the Sun-3 and `0xFFF00000` to `0xFFFFFFFF` for the Sun-4). Both Sun-3 and Sun-4 DVMA hardware uses supervisor access for checking protection.

The driver writer must account for these mappings, as should be evident from the diagram below.

Figure 2-5 *System DVMA*


DMA Device
VME BUS
Slave Decoder
If Address in Low megabyte, Map it to High megabyte
CPU
MMU
Buffer
OnBoard Mem
On-Board Bus Masters (like the Ethernet chip)

Devices can only make DVMA transfers in memory buffers which are from (or redundantly mapped into — see below) the low-memory areas reserved as DVMA space. The memory-management hardware will then recognize references to these areas and map them into the high megabyte of system virtual address space, an area known as DVMA space. Likewise, if a driver needs to allocate space for a DMA transfer, it must do so by way of a mechanism that guarantees its allocation from DVMA space. There are several ways of making this guarantee:

- `rmalloc()` can be used with the `iopbmap` argument. This will get a small block of memory from the beginning of the DVMA space. Such small blocks of memory are usually used for control information, and not for large blocks of data.

- For a large buffer, the driver can statically declare a `buf` structure (which is a buffer header that contains a pointer to the data) and then use `mbsetup()` to allocate a buffer for it from DVMA space. This mechanism is primarily intended for block devices but is perfectly adaptable for use by character devices that need large DMA buffers.

When dealing with addresses which are in DVMA space, the driver must strip off the high bits by subtracting the external variable `DVMA`, which contains the address of DVMA (declared as an array of characters). `DVMA` is initialized by the system to either `0xF00000` (for Sun-2s) or `0xFF00000` (for Sun-3s and Sun-4s). If the driver fails to make this adjustment, the device will attempt to use a null address — in the high megabyte — and the CPU board will not respond to it.

NOTE *Addresses received by way of* `mbsetup()` *(and* `MBI_ADDR()`*) do not have to be adjusted in this fashion, as* `mbsetup()` *will have already adjusted them to be relative to the start of DVMA space.*

When the device, in turn, uses the address, the address reference comes down the bus and through a slave decoder, which adds the machine-specific offset to it to map it back into the high megabyte of system virtual memory.

Sun DMA is called DVMA because the addresses which the device uses to communicate with the kernel are virtual addresses like any others. The driver, as part of the kernel, is privy to implementation dependent information, and knows that it must chop off the high-bits of any address intended for the device. This allows the MMU to recognize the addresses destined for the Main Bus and to act accordingly. The device, however, knows nothing of this except that its buffers are mapped to the high megabyte of system virtual memory.

User processes, it should be noted, cannot do DVMA directly into their own address spaces. The kernel, however, provides a way of getting around this limitation by supporting the redundant mapping of physical memory pages into multiple virtual addresses. In this way, a page of user memory (or, for that matter, a page of kernel memory) can be mapped into DVMA space in such a way that transferred data immediately appears in (or immediately comes from) the address space of the process requesting the I/O operation. All that a driver need do to support such direct user-space DVMA is to set up the kernel page maps with the routine `mbsetup()` — the details of the mapping will then be automatically handled by the kernel.

If you wish to do DMA over the Main Bus, you must make the appropriate entries in the kernel memory map. There are two functions, `mbsetup()` and `mbrelse()`, to help with this chore.

DMA on ATbus Machines

The Sun386i uses the Intel 80386 chip. This chip has an integrated MMU, so the I/O devices cannot access the Sun MMU address-translation facility and therefore must use physical addresses to access memory directly.

To do DMA on the Sun386i, you must make certain changes in the kernel's memory map (its page tables). Use the `mbsetup()`, `dma_setup()`, `mbrelse()`, and `dma_done()` routines to make these changes. The changes you must make to the kernel memory map are described with these routines in the *Kernel Support Routines* appendix.

3

Overall Kernel Context

Overall Kernel Context **41**

3.1. The System Kernel 41
3.2. Devices as ''Special'' Files 42
3.3. Run-Time Data Structures 47
  The Bus-Resource Interface 49
  Autoconfiguration-Related Declarations 55
  Other Kernel/Driver Interfaces 56

3

Overall Kernel Context

3.1. The System Kernel

Device drivers are parts of the SunOS kernel, a fact that must be appreciated to understand the ways in which drivers differ from user-level programs. The kernel is the crucial system program responsible for the control and allocation of system resources, including the processor, primary memory and the I/O devices. In most ways it's just like any user program, being a more or less cleverly constructed structure shaped to its particular goals. In other ways, however, it's significantly different from a user program:

- For one thing, the kernel is thick with the details of hardware implementation and function. This tends *not* to be true of user programs, precisely because the kernel shields them from the need to consider device-specific details.
- For another, the kernel (and thus its drivers) runs in supervisor mode. This means that drivers can often perform privileged device operations that can't be performed by user processes, even if those processes have access to the necessary device registers.
- The kernel memory context is not entirely paged. Certain parts of the kernel are paged, but drivers can safely assume that their text and data are resident and stationary within physical memory.
- Programmers of ordinary user processes rarely need to concern themselves with physical addresses and virtual-to-physical address mappings. Device-driver developers, however, deal simultaneously with user virtual addresses, kernel virtual addresses and physical bus addresses. Special functions (see the *Kernel Support Routines* appendix) are provided to help drivers with the various address mappings they're called upon to perform.
- Finally, the kernel provides a far different external interface than do user processes. It's possible for user processes to communicate with and dispatch tasks to other user processes by way of system inter-process communications mechanisms (like signals and pipes) but to do so they must first make special arrangements with those other processes. The kernel, on the other hand, exists to provide services to user processes and it provides a special mechanism — the system call — by which user processes can call upon it to do so. This is not to say that user processes and the kernel (that is, the drivers) can't also use system inter-process communications mechanisms like signals. It's certainly possible, for example, to write a driver so that it will send a signal to a user process as part of its handling of a specified event. However, in the

norm, user processes and the kernel communicate by way of system calls.

NOTE *On all Sun systems, system calls are defined in* `/usr/sys/os/init_sysent.c`, *which users may edit to add system calls. This file is provided with all Sun-2, Sun-3, Sun-4, and Sun386i systems.*

System calls can, for all intents and purposes, be understood as calls by user processes to kernel subroutines; they involve, however, far more profound system state changes that do regular subroutine calls. When system calls are processed, the processor is placed in supervisor state (and, in Sun-2 systems, the kernel virtual address space becomes current in place of the the user virtual address space). The user process is suspended and the kernel begins to run, but since it runs on behalf of that user process which issued the system call, it can be viewed as that user process continuing execution in kernel mode. Such "kernel-mode" processes continue to run (with pauses whenever they sleep or yield to a higher-priority process) until the system call processing is completed. At this time the scheduler is called to choose the next user process to be dispatched.

Some system calls can be completely processed without calling any device driver routines. The system call `lseek()` is in this class, it requires only that a software file position indicator be reset. Like many system calls — those related to process control, inter-process communication, timing services, and status information — it can be handled entirely in software. Requests for I/O, however, usually involve some action on the part of a peripheral device. In this case the kernel calls (through a branch table mechanism described below) a routine within the I/O device's driver. The driver will then initiate the I/O operation and, if necessary, `sleep()` until the data is available; in the meantime the kernel will dispatch another user process.

3.2. Devices as "Special" Files

When a user process issues a system call, execution shifts to the kernel. Then, for I/O-related system calls, the kernel distinguishes requests related to regular named files (that is, files on a block device like a disk) from requests related to other kinds of I/O devices (like terminals or printers). In the interests of uniformity, these devices are viewed as "special" files which (by convention) are collected in the */dev* directory. These special files are not created in the usual way. The information in their *i-nodes* (the system structures that define the state of files) is quite different from the information maintained for regular files, and, as a consequence, special files can only be created with the `mknod` (make a node) administration command. Instead of the addresses that will locate the contents of a regular file on a disk, the i-nodes of special files (devices) contain the information necessary to determine the corresponding device driver (the major device number), the device class (block or character), and the minor device number.

When a file of any type is accessed, the kernel needs to determine which device driver is responsible for it. To make this determination, it must get the name of the device associated with the file. From that name it can derive (using a device-independent kernel subsystem) an i-node and thus a major device number (as well as a minor device number and a device class).

The connection between the device name and its major number is made by way of the device entry in the */dev* directory (more specifically, by way of the i-node information associated with the device entry). The i-node for a device special file contains a major device number, which is used to index one of the two *device switches*. These switches, `bdevsw` (the block device switch) and `cdevsw` (the character device switch) are actually arrays of structures, and the major device number selects a driver by indexing one of these structures. (The minor device number is then passed to the driver for local interpretation).

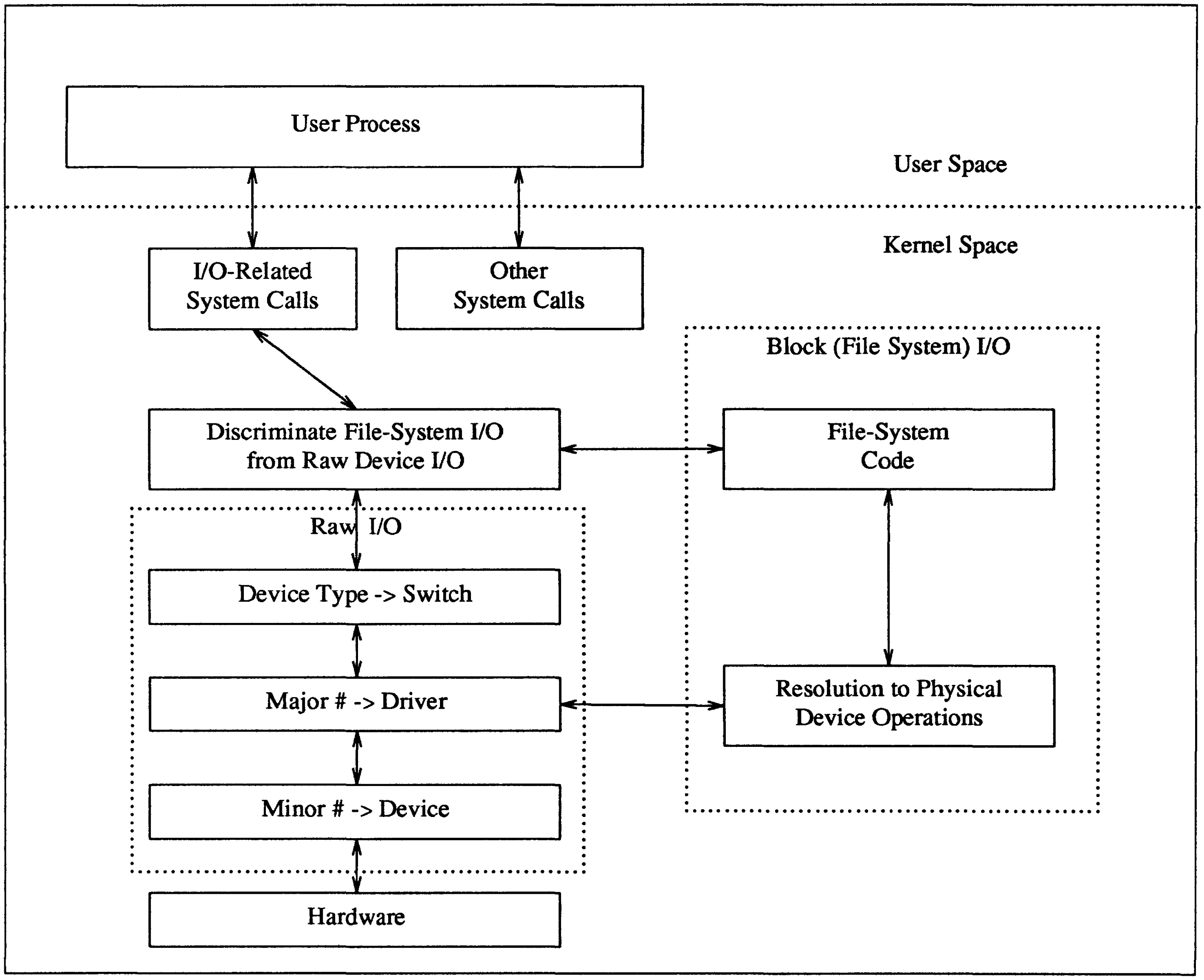
Using the `ls -l` command on the `/dev` directory shows you the i-node information associated with special files:

Table 3-1 *A Sample Listing of the /dev Directory*

Type	*permissions*	*size*	*owner*	*major #*	*minor #*	*date*	*name*
c	rw--w--w-	1	henry	0,	0	Feb 21 09:45	console
c	rw-r--r--	1	root	3,	1	Dec 28 16:18	kmem
c	rw-------	1	root	3,	4	Jan 13 23:07	mbio
c	rw-------	1	root	3,	3	Jan 13 23:07	mbmem
c	rw-r--r--	1	root	3,	0	Dec 28 16:18	mem
c	rw-rw-rw-	1	root	13,	0	Dec 28 16:18	mouse
c	rw-rw-rw-	1	root	3,	2	Feb 22 16:40	null
c	rw-------	1	root	9,	0	Dec 28 16:19	rxy0a
c	rw-------	1	root	9,	1	Dec 28 16:19	rxy0b
							.
							.
							.
c	rw-------	1	root	9,	6	Feb 25 1984	rxy0g
c	rw-------	1	root	9,	7	Dec 28 16:19	rxy0h
b	rw-------	1	root	3,	0	Feb 25 1984	xy0a
b	rw-------	1	root	3,	1	Jan 17 20:12	xy0b
							.
							.
							.
b	rw-------	1	root	3,	6	Dec 28 16:19	xy0g
b	rw-------	1	root	3,	7	Dec 28 16:19	xy0h

When a user process wishes access to a system service, it makes a system call. The subsequent flow of control looks somewhat like this:

Figure 3-1 *I/O Paths in the UNIX system*


User Process
User Space
Kernel Space
I/O-Related System Calls
Other System Calls
Block (File System) I/O
Discriminate File-System I/O from Raw Device I/O
File-System Code
Raw I/O
Device Type -> Switch
Major # -> Driver
Resolution to Physical Device Operations
Minor # -> Device
Hardware

When you add a new device driver you must add entries to one or both of the device switches. Since we are discussing only character-oriented devices in this manual, we will ignore the `bdevsw` structure and concentrate on the `cdevsw` structure. But note that it's common for drivers to appear in both tables; this happens because block-devices almost always support raw character I/O.

Application programs make calls upon the operating system to perform services such as opening a file, closing a file, reading data from a file, writing data to a file, and other operations that are done in terms of the file interface. The operating system code turns these requests into specific requests to the device driver involved with that particular file. The glue between the specific file operation involved and the device driver entry-point is through the `bdevsw` and `cdevsw` tables.

Each entry in `bdevsw` or `cdevsw` contains pointers to a driver's entry-point functions. The position of an entry in the structure corresponds to the major device number assigned to the device. The minor device number is passed to the device driver as an argument. Usually, the driver uses it to access one of several identical physical devices, but it is also possible for it to be encoded so that multiple minor numbers indicate the same device, but different operating modes. For example, one minor number might indicate a specific tape device, as well as the fact that the device is to be rewound when being closed, while another indicates the same device without the rewind. A minor number may also indicate a controller/device pair. Such breadth of interpretation is possible because the minor number has no significance other than that attributed to it by the driver itself.

The `cdevsw` table specifies the interface routines present for character devices. Each character device may provide seven functions: *xx*`open()`, *xx*`close()`, *xx*`read()`, *xx*`write()`, *xx*`ioctl()`, *xx*`select()`, and *xx*`mmap()`. (While character drivers sometimes have "strategy" routines, this name is simply a carryover from the world of block drivers, and `cdevsw` thus has no *xx*`strategy()` entry point). If you wish calls on a routine to be ignored — for example *xx*`open()` calls on non-exclusive devices that require no setup — the `cdevsw` entry for that driver can be given as `nulldev`; if a call should be considered an error — for example *xx*`write()` on read-only devices — `nodev`, which returns immediately with an error code, can be used. For terminals, the `cdevsw` structure also contains a pointer to an array of `tty` structures associated with the driver.

Note: the device switch tables do not include pointers to the driver initialization and interrupt handler functions. Pointers to these functions appear in separate mbvar structures (discussed below).

Here's what the declaration of an entry in the character device switch looks like. Each entry (row) is the only link between the main SunOS code and the driver. The declaration and initialization of the device switches is in `/usr/sys/sun/conf.c`:

```
struct cdevsw {
    int  (*d_open)();       /*  routine to call to open the device    */
    int  (*d_close)();      /*  routine to call to close the device   */
    int  (*d_read)();       /*  routine to call to read from the device   */
    int  (*d_write)();      /*  routine to call to write to the device    */
    int  (*d_ioctl)();      /*  special interface routine    */
    int  (*d_stop)();       /*  flow control in tty's  */
    int  (*d_reset)();      /*  reset device and recycle its bus resources  */
    struct tty *d_ttys;     /*  tty structure     */
    int  (*d_select)();     /*  routine to call to select the device    */
    int  (*d_mmap)();       /*  routine to call to mmap the device     */
    struct streamtab *d_str; /* support for STREAMS  */
};
```

Only teletype-like devices (such as the the console driver, the `mti` driver, and the `zs` driver) use the `tty` structure. All other devices set it to zero.

Routines in the kernel call specific driver routines indirectly by way of the table with the major device number. A typical kernel call to a driver routine will look something like:

```
    (*cdevsw[major(dev)].d_open) (params. . .);
```

And here is a typical line from `/usr/sys/sun/conf.c`, which initializes the requisite pointers in the `cdevsw` structure:

```
          .
          .
          .
          All the other cdevsw entries between 0 and 13 appear first
    {
cgoneopen,  cgoneclose, nodev,   nodev,   /*14*/
cgoneioctl, nodev,   nodev,   0,
seltrue,    cgonemmap,
    },

          Then all the other cdevsw entries from 15 up
          .
          .
          .
```

In the Sun system, a number of devices in `cdevsw` are preassigned. The table below shows *some* of these assignments at the time of this writing. It is not complete, and besides, new devices are always being added. In allocating a major number to the new device which you're installing, make sure that you don't choose one that's already been allocated. `/usr/sys/sun/conf.c` will give the major device numbers as currently allocated on your system. Choose yours so it will go at the end.

Table 3-2 *Current Major Device Number Assignments*

Major Device Number	*Device Abbreviation*	*Device Description*
0	`cn`	Sun Console
1	*Not Available*	No Device
2	`sy`	Indirect TTY
3	*Memory special files*	
4	*Not Available*	No Device
5	`tm`	Raw Tapemaster Tape Device
6	`vp`	Ikon Versatec Parallel Controller
7	*Not Available*	No Device
8	`ar`	Archive Tape Controller
9	`xy`	Raw Xylogics Disk Device

Table 3-2 *Current Major Device Number Assignments— Continued*

Major Device Number	*Device Abbreviation*	*Device Description*
10	`mti`	Systech MTI
11	`des`	DES Chip
12	`zs`	UARTS
13	`ms`	Mouse
14	`cgone`	Sun-1 Color Graphics Board
15	`win`	Window Pseudo Device
16	*Not Available*	Log Device
17	`sd`	Raw SCSI disk
18	`st`	Raw SCSI tape
19	*Not Available*	No Device
20	`pts`	Pseudo TTY
21	`ptc`	Pseudo TTY
22	`fb`	Sun Console Frame Buffer
23	`ropc`	RasterOp Chip
24	`sky`	SKY Floating Point Board
25	`pi`	Parallel input device
26	`bwone`	Sun1 Monochrome frame buffer
27	`bwtwo`	Sun-2 Monochrome frame buffer
28	`vpc`	Parallel Driver for Versatec printer
29	`kbd`	Sun Console Keyboard Driver
30	`xt`	Raw Xylogics 472 Tape Controller
31	`cgtwo`	Sun-2 Color Frame Buffer
32	`gpone`	Graphics Processor
33	`sf`	Raw SCSI Floppy
34	`fpa`	Floating-Point Accelerator
35	*Not Available*	STREAMS Support
36	*Not Available*	No Device
37	*Not Available*	STREAMS Clone
38	`pc`	Sun PC Driver
39	`cgfour`	Sun-3/110 Color Frame Buffer
40	*Not Available*	STREAMS NIT
41	*Not Available*	Dump Device
42	`xd`	Xylogics 7053 SMD Disk Driver
	. . .	
	. . .	
	. . .	

3.3. Run-Time Data Structures

If you skip ahead and read the chapter on *Configuring the Kernel* you will see a discussion of the procedures by which Sun systems are reconfigured to include new devices and drivers. There are two major programs involved in this process. The first is `config`, which reads the kernel config file and generates the data-structure tables which specify the configuration of the new kernel. You will also note, in that chapter, references to the kernel's autoconfiguration process (sometimes called `autoconfig`). The autoconfiguration process verifies that the devices specified in the config file are actually installed and working, and adjusts the kernel data structures accordingly.

The autoconfiguration approach was first introduced in 4.1BSD as part of a larger kernel rationalization, and it significantly increases the flexibility of the kernel configuration process, for example, by allowing multiple device controllers to be driven by the same instance of a driver.

The autoconfiguration process is called by the kernel during its boot-time initialization. It does several things:

- It verifies that the information in the kernel config file is correct; that is to say, it verifies that the devices which the kernel thinks are installed are actually installed. It does this by calling device-specific *xx*`probe()` routines that are supplied by the driver.
- It completes the initialization of the kernel data structures that were declared by `config` and linked into the kernel by way of `ioconf.c` (a file which `config` creates but cannot fully initialize). These structures, which are defined in `<sundev/mbvar.h>` and shall hereafter be known as the *mbvar structures*, form a good part of the run-time environment of the driver routines.
- It maps the device registers (or memory) into kernel virtual space.
- It sets up polling interrupts on Multibus systems.

The autoconfiguration code does its work, as its name indicates, without worrying the driver developer too much. It's only necessary for the developer to know what conventions to follow and what options exist. The rest will take care of itself.

Note: readers who have written only System V drivers will perhaps find this all a bit mysterious. In System V, as in BSD UNIX systems, the driver interface to the kernel is defined primarily by the function switch (either `cdevsw` *or* `bdevsw`*) by which driver routines are called, by the parameters these routines are passed and by the values they return. So far so good, but then there are the differences. In System V drivers, nothing like the mbvar structures exists, and generic kernel structures (like the* `user` *structure) are used far more heavily than in 4.2BSD, where mbvar-like structures are consulted by preference. Sun's operating system is, of course, derived from 4.2BSD, and its driver interface is quite similar.*

The "mb" in the name of the *mbvar* structures clearly recalls the primary motivation of the kernel rewrite in which they were introduced — to improve the management of bus resources. The "mb" is derived from the initials of the *Multibus*, around which older generation Sun machines were built. Newer machines, while built around the VMEbus, nevertheless continue to bear the traces of the past in these *mbvar* structure names, names which are now taken to stand for "Main Bus" rather than for "Multibus."

During the configuration of the kernel, an edifice is built of the *mbvar* structures and its initialization is begun. The edifice consists of a structure which represents the bus itself, two arrays of structures (one representing system controllers; the other, devices) and a number of inter-structure field-to-field links of various kinds.[3] The details of the edifice depend upon the information in the kernel config file, and upon the compile-time declarations made by the individual drivers. During boot time, the initialization that `config` began is completed by the autoconfiguration process.

Then, at run time, the *mbvar* structures are used by both the drivers and the kernel to manage the bus and its interaction with the devices. The *mbvar* structures are linked to each other in quite a complex fashion, for device characteristics and thus device driver structures vary greatly, and these structures are intended to support a great variety of access paths: device to controller, device to driver, controller to driver, and so on. Driver developers do not, however, need to concern themselves with the details of the inter-structure links and access paths. Driver routines will be called by the kernel with pointers to the *mbvar* structures of interest to them. They are then free to build that information into whatever local structures they find most convenient for the representation of whatever access paths are of interest to them.

So, to sum up, the Sun kernel/driver runtime interface can be seen as being built in two different sections. One of these sections is composed of the *mbvar* structures, constructed into interlinked arrays to represent specific kernel configurations on specific machines. The other is similar to the generic SunOS kernel/driver interface, consisting as it does of the two device switches, the `user` and `proc` structures, parameter conventions and a few miscellaneous variables. We will now discuss the details to these two interfaces.

The Bus-Resource Interface

All controllers are installed on the main system bus, and all slave devices (like disks and tape drivers) are attached to their controllers.[4] Additionally, each controller is associated with a device driver, which is really a controller driver. The *mbvar* data structures reflect these relationships, not only in terms of the fields that they contain but in terms of the ways these fields are linked together.

The following *mbvar* structure fields are the ones most relevant to driver developers.

mb_hd The first data structure, `mb_hd`, is the Main Bus header data structure. There is only one such structure, for Sun systems have only one Main Bus. It contains a queue of `mb_ctlr` structures, each one representing a controller waiting for DVMA space. The queue only contains entries when DVMA space is full. It also contains other bus-status information. For example, if a driver has exclusive access to the bus, this is noted in `mb_hd`. Device drivers never directly access the fields in `mb_hd`.

---

[3] It's not always clear just when a device is a "controller", and when it's a "device". The extreme cases are clear: if a device attaches to the bus, fields interrupts and has other, so-called "slave" devices, then it's a controller. Likewise, if a device attaches to a controller rather than to the bus, it's a slave device. The confusion surrounds devices which attach to the device and field interrupts, but which do *not* have slave devices. Such "devices" would, in many ways, be better thought of as "controllers" which control only themselves.

[4] Sometimes, in this manual, the word "device" will be used in a generic sense to denote either a "free" device that attaches directly to the system bus rather than to a separate controller, or a regular slave device. This generic usage occurs, for example, whenever the term "device driver" is used — such programs would more accurately be described as "controller drivers". In this section, however, we're being extremely precise — free devices attach to the system bus, and so they're called "controllers", not "devices".

mb_ctlr Each slave-device controller on the Main Bus has an `mb_ctlr` structure associated with it. (This structure contains all of the configuration-dependent information which the kernel needs in interactions with the controller's driver, as well as some status information. It is `mb_ctlr` that is queued onto `mb_hd` during a wait for DVMA space. The following fields within `mb_ctlr` are of interest even for character devices (there are others that are used only by block devices):

mc_ctlr
The controller index for the corresponding controller, for example, the '0' in `sc0`. Used to index into arrays of driver-specific controller status and control structures.

mc_addr
The address of the controller (control and status registers and RAM) in bus space.

mc_dmachan

On the Sun386i only, a field containing the DMA channel.

mc_space
A bit pattern which identifies the address space within which the controller is installed.

mc_intpri
The interrupt priority level of the controller. This is to be given in the config file and should be used, in the driver source, only as an argument to `spln()` — e.g. `splx(pritospl(mc_intpri))`.

mc_intr
On Sun-2, Sun-3, and Sun-4 systems, pointer to the `vec` structure that specifies vectored interrupt behavior (or `NULL` if vectored interrupts are not used). If `mc_intr` is set, then the fields within the `vec` structure become significant:

v_func
Pointer to the vector-interrupt function.

v_vec
Vector number associated with the function in `v_func`.

v_vptr
A pointer to the 32-bit argument to be passed to the driver vector-interrupt routine. Defaults to the controller number of the interrupting device, though it can be reset within the driver. It's often set by the driver *xx*`attach()` routine to contain a local structure pointer. On the Sun386i system, this field contains the irq (interrupt

request channel). The Sun386i system does not support vectored interrupts, so the `v_*` fields are not present.

`mc_alive`
Set to one by the autoconfiguration process if the controller is determined to be present. Otherwise left at 0.

`mc_mbinfo`
Main Bus resource allocation information (Used by `MBI_ADDR()`, `mbsetup()` and `mbrelse()`).

`mb_device` "Free" devices (devices with no separate controllers) as well as "slave" devices, are represented to the kernel bus-management routines by an instance of the `mb_device` structure. (This is as it has been since 4.1BSD, but it's not ideal — if free devices were taken as controllers and represented by an `mb_ctlr` structure, then `mb_device` would only be for slave devices and would contain fewer fields). `mb_ctlr` contains all of the configuration-related data for the free or slave device. If a controller has multiple slave devices attached to it, there will be as many `mb_device` structures associated with its `mb_ctlr` structure. The following fields within `mb_device` (which are set by the configuration system and are not normally reset by the driver) are of interest:

`md_driver`
A pointer to the `mb_driver` structure associated with this device.

`md_unit`
The device index for the corresponding device, for example, the '0' in `xy0`. Used to index into arrays of driver-specific device status and control structures.

`md_slave`
The slave number of the device on its controller.

`md_addr`
The base address of the device (its control/status registers and perhaps some RAM). For most Multibus devices, this will be an address in I/O space, though for memory-mapped devices this will be an address in Memory space. For VMEbus machines, it's the particular address space within which the device is attached. Unused for devices on controllers.

`md_dmachan`
On the Sun386i only, a field containing the DMA channel.

`md_intpri`
The Main Bus priority level of the device (the priority that is passed to `pritospl()`). Used to parameterize the setting of hardware priorities. Unused for devices on controllers.

md_intr
On Sun-2, Sun-3, and Sun-4 systems, pointer to the `vec` structure that specifies vectored interrupt behavior (or `NULL` if vectored interrupts are not used). Unused for devices on controllers. On the Sun386i system, this field contains the interrupt channel.

md_flags
The optional `flags` parameter from the system config file is copied to this field, to be interpreted by the driver. *Only the driver uses the information in this field.* If `flags` was not specified in the config file, then this field will contain a 0.

md_alive
Set by the autoconfiguration process to 1 if *xx*`probe()` finds the device, otherwise it's left at 0. Incidently, if *xx*`probe()` fails to find the device, the autoconfiguration process will also leave the device position in the *xx*`dinfo()` array (if the driver has one) at 0. The driver is free to test either variable (in its *xx*`open()` routine) to determine *xx*`probe()`'s verdict.

mb_driver The system assumes that the source code of your driver declares a `mb_driver` structure named *xx*`driver()`. This structure contains information relevant to the device driver *as a whole,* as opposed to information about individual devices or controllers. It differs in several important manners from the device and controller structures. For one thing, it contains a number of pointers to driver functions. These pointers, like those in `cdevsw` and `bdevsw`, are used by the kernel as entry points into the driver. For another, it's initialized not by the configuration system, but within the driver source code itself — if fact, several of the routines in *xx*`driver()` are actually called by the kernel autoconfiguration process to complete the driver-related kernel initialization. *(Note: while the driver has responsibility for initializing the fields in* `xxdriver`*, it is still limited, at run time, to reading these fields — it cannot ever change them).*

xx`driver()` must be known more intimately by the driver developer than either the driver `md_ctlr` structure or the driver `md_device` structure. We will therefore give its complete definition:

```
struct mb_driver {
    int      (*mdr_probe)();         /* check device/controller installation */
    int      (*mdr_slave)();         /* check slave device installation */
    int      (*mdr_attach)();        /* boot-time device initialization */
    int      (*mdr_go)();            /* routine to start transfer */
    int      (*mdr_done)();          /* routine to finish transfer */
    int      (*mdr_intr)();          /* polling interrupt routine */
    int      mdr_size;               /* amount of memory space needed */
    char     *mdr_dname;             /* name of a device */
    struct   mb_device **mdr_dinfo;  /* backpointers to mbdinit structs */
    char     *mdr_cname;             /* name of a controller */
    struct   mb_ctlr **mdr_cinfo;    /* backpointers to mbcinit structs */
    short    mdr_flags;              /* want exclusive use of Main Bus */
    struct   mb_driver *mdr_link;    /* interrupt routine linked list */
};
```

Here is a brief discussion of the fields in the `mb_driver` structure that you will need to initialize when declaring *xx*`driver()`. Note that many of the fields in `mb_driver` are for the use of block drivers only — they're presented here as useful background information.

mdr_probe

is a pointer to the driver *xx*`probe()` routine. *xx*`probe()` is called for every controller and every independent device (with no separate controller) given in the kernel config file. *xx*`probe()` determines if the device/controller is actually installed. If it is, it returns the amount of bus space consumed by the device/controller to the autoconfiguration process, where this space is then mapped into system address space. When *xx*`probe()` fails, it returns 0.

mdr_slave

is a pointer to an *xx*`slave()` function within your driver. *xx*`slave()` is analogous to *xx*`probe()`, and serves the same function for devices which are driven by separate controllers. Unlike *xx*`probe()`, however, *xx*`slave()` exists only for controllers that may have multiple devices — it's therefore quite rare in character device drivers.

mdr_attach

is a pointer to an *xx*`attach()` function within your driver. *xx*`attach()` is called during the autoconfiguration process, where it does preliminary setup and initialization for a device or controller. It's commonly used within disk and tape drivers to perform setup tasks like the reading of labels, and in character drivers for tasks like initializing interrupt vectors and reserving blocks of memory. Initialize this field only if there's an *xx*`attach()` routine in your driver.

mdr_go
mdr_done

are pointers to *xx*`go()` and *xx*`done()` functions within the driver. These functions usually don't exist for character drivers, and these fields are consequently 0.

mdr_intr

is a pointer to a polling interrupt routine within your driver. Such a polling routine is used for the "auto-vectoring" of interrupts in systems where the interrupt "vector" can only be based on the interrupt priority. This is the case on all Multibus machines, and if there's any chance that your driver will someday be run on a Multibus machine you should include a polling interrupt routine and plug it in here.

If you have a Sun source license, and take the opportunity it affords to examine a number of drivers, you may note an inconsistency in the naming of interrupt routines:

- Some drivers have two interrupt routines: a polling interrupt routine named *xx*`poll()` and a vector interrupt routine, named *xx*`intr()`. In such cases *xx*`poll()` determines the unit number of the interrupting device and then calls *xx*`intr()` to actually handle the interrupt.
- Other drivers have only one interrupt routine. The routine is named *xx*`intr()` and called from `mdr_intr`, but it nevertheless contains polling code. This, like the naming of the field `mdr_intr` (which really should be `mdr_poll`) is an artifact of early Sun systems, in which drivers were written for the Multibus only — in these systems *xx*`intr()` was *the* interrupt routine, and it always contained polling code.

In any case, remember that any routine called from `mdr_intr` must check the polling chain, regardless of its name. If you will not support Multibus machines, and thus need no polling interrupt routine, put a zero in this field.

mdr_size

is the size — in bytes — of the memory required for the device. This field is initialized with a value identical to that which *xx*`probe()` returns upon success, and specifies the amount of space that needs to be mapped into system memory by the autoconfiguration code. The value returned by *xx*`probe()`, while identical, is used only to indicate if the device was found.

mdr_dname

is the name of the device for which this driver is written.

mdr_dinfo

is a pointer to a pointer to the `mb_device` structure in *xx*`dinfo()`. This pointer is filled in during autoconfiguration (see section below on *Autoconfiguration-Related Declarations*) and is necessary to work back from the device unit number to the correct `mb_device` structure by way of an index operation.

mdr_cname

is the name of the controller supported by this driver (for example, *sc* supports the controllers `sc0`, `sc1`, etc). This field takes the form of a regular null-terminated C string. Fill it in if you actually have a controller.

mdr_cinfo

is a pointer to a pointer to an `mb_ctlr` structure declared in the driver. This pointer is filled in during autoconfiguration (see the section below on *Autoconfiguration-Related Declarations*) and is necessary to work back from the device unit number to the correct `mb_ctlr` structure by way of an index operation.

mdr_flags

consists of some flags, as follows:

`MDR_XCLU`

The device needs exclusive use of Main Bus while running. This flag is used only by `mbgo()` and `mbdone()` routines (which are not documented in this manual), and it guarantees exclusive use only among drivers which use it to enforce an exclusive-use protocol. Not all drivers do so.

`MDR_BIODMA`

For devices that do DMA on the Main Bus (such drivers call `mbgo()` and `mbdone()`). This flag tells the kernel that it must lock other DMA devices off the bus.

`MDR_DMA`

For devices which use DMA, either to transfer large blocks of data or simply to transfer small blocks of control information. The drivers for such devices call `mbsetup()`. This flag tells the kernel that it must lock other DMA devices off the bus, and all DMA drivers should set it.

`MDR_SWAB`

I/O buffers are to be `swab()`'ed — that is, pairs of data bytes are to be exchanged. This flag is used to push the `swab()` out of `mbgo()` and `mbdone()` and down into the Main Bus driver.

`MDR_OBIO`

The device is installed in on-board I/O space.

Of these, `MDR_XCLU`, `MDR_SWAB` and `MDR_OBIO` are potentially to be used for user character devices. These flags must be OR'ed together if you wish to place any of that information there. Place a zero (0) in this field if none of the flags apply to your driver.

mdr_link

This field is used by the autoconfiguration routines and is not for the driver's use.

Autoconfiguration-Related Declarations

At the top of each driver, after the include statements, is a group of declarations that are used by the autoconfiguration process to finish the initialization of the *mbvar* structures. Here, as an example, are the relevant declarations from the Sky Floating-Point Driver:

```
/* Driver Declarations for Autoconfiguration    */
int skyprobe(), skyattach(), skyintr();
struct  mb_device *skyinfo[1];  /* Only Supports One Board */
struct  mb_driver skydriver = {
    skyprobe, 0, skyattach, 0, 0, skyintr,
    2 * SKYPGSIZE, "sky", skyinfo, 0, 0, 0,
};
```

The first line declares the names of the autoconfiguration-related entry point routines for the driver. In this case there are only three — `skyprobe()`, `skyattach()` and `skyintr()`. These declarations are necessary because, in a few lines, we will use the names to initialize the driver's `mb_driver` structure.

The second line declares an array (in this case of dimension one) of pointers to `mb_device` structures. By the time the driver is linked into the kernel, `config` will have already declared an array of `mb_device` structures that contains an entry for each of the devices named in the kernel config file. When the kernel is booted, the autoconfiguration process initializes each driver's *xx*`info()` array to indicate the `mb_device` structures corresponding to its devices, with each device's unit number being used as its subscript into the *xx*`info()` array. The Sky driver is slightly atypical in that it only supports one device; normally the device count provided by `config` in a macro "NXX" (which is set to the number of devices noted in the config file) would be the subscript in this declaration.

If this was a driver for a controller with slave devices, the second line would be followed by an analogous one that declared an array of pointers to `mb_ctlr` structures.

The third line both declares and initializes the `mb_driver` structure that represents this driver. The fields within the structure are described in detail in the previous section.

Other Kernel/Driver Interfaces

The kernel/driver interface is almost entirely contained within the *mbvar* structures and the parameter conventions of the driver routines. There are, however, a few other common kernel/driver interface points, which are given in this section.

WARNING

The `user` ***structure is valid for the current process only while execution is in the top half of the driver. It must never be accessed from the bottom half.***

The kernel `user` structure contains a few fields of interest to drivers. This structure, which maintains status information for the current user process (and which is swapped in and out with the process it describes), is used far less by Sun drivers than it is by System V drivers. This is because, in SunOS, the `user` structure does not define the address of the characters to be written (or the place for characters to be read to). The Sun kernel uses `uio` structures for this purpose, and passes them as parameters to the driver *xx*`read()` and *xx*`write()` routines. (See *Some Notes About the UIO Structure* in the *The "Skeleton" Character Device Driver* chapter of this manual).

Still, three fields within the `user` structure remain of interest to device drivers. They are:

`u.u_qsave`
is a `setjmp()` environment buffer that can be used to save the current stack in preparation for a possible `longjmp()` return. `setjmp()` and `longjmp()` are useful in drivers that need to intercept signals, and then to wake sleeping processes. They can also be used for error handling. For more information, see the `setjmp(3)` man page.

`u.u_error`
If an I/O operation is not successful, the driver must return an error code (defined in `<errno.h>`), which is plugged into `u.u_error`. From here it's normally stored in the per-process global variable `errno` in the user context. (Note that in most cases the *kernel* plugs the value into `u.u_error`, and it is not necessary for the driver to do so. In fact, a driver cannot access `u.u_error` in its interrupt routine, where transfer errors are normally detected, since the current `user` structure is unlikely to belong to the process for which the failed I/O was being performed).

`u.u_procp`
The `u.u_procp` field in the `user` structure is a pointer to the processs (`proc`) structure for the current process. The `proc` structure contains the information that the system needs about a process even when it is swapped out. `u.u_procp` is used by drivers which contain `select()` routines. See the *Variation with "Asynchronous I/O" Support* section of the *The "Skeleton" Character Device Driver* chapter of this manual for details.

Drivers may occasionally need to know what kind of machine they're running on. They can find out by querying a variable, `cpu`, which, while not in the `user` structure, is available to them by including `../machine/cpu.h`. This variable is initialized by the kernel on the basis of information in the ID PROM, and is set to one of the following values:

```
CPU_SUN2_50
CPU_SUN2_120
CPU_SUN3_50
CPU_SUN3_110
CPU_SUN3_160
CPU_SUN3_260
CPU_SUN4_260
CPU_I386_AT386
```

Note that when compiling for a Sun-2 system, only the Sun-2 names are available; likewise for Sun-3s, Sun-4s and Sun386i's.

Related to the `CPU_SUNX_XX` names are the `SUNX_XX` ifdefs. These are set at compile time on the basis of information in the config file, and can be used to eliminate code or data that is unnecessary for machines of any particular type. In general, it's possible (and advised) to write drivers that can compile and run on a variety of Sun machines with no changes.

DVMA drivers will often need to know the address of kernel DVMA space on the host machine (See the *Sun Main-Bus DVMA section in the Hardware Context* *chapter) so that they can subtract it from system virtual addresses to get addresses relative to the start of DVMA space. The external variable* `DVMA`, *declared as an array of characters, is available for this purpose.*

The external variable `hz` gives the number of clock ticks per second on the host system.

The external variable `KERNELBASE` gives the start of kernel address spece in the current memory context.

4

Kernel Topics and Device Drivers

Kernel Topics and Device Drivers **61**

4.1. Overall Layout of a Character Device Driver 61
4.2. User Space versus Kernel Space 63
4.3. User Context and Interrupt Context 63
4.4. Device Interrupts 64
4.5. Interrupt Levels 65
4.6. Vectored Interrupts and Polling Interrupts 66
4.7. Some Common Service Routines 69
  Timeout Mechanisms 69
  Sleep and Wakeup Mechanism 69
  Raising and Lowering Processor Priorities 70
  Main Bus Resource Management Routines 71
  Data-Transfer Functions 71
  Kernel **`printf()`** Function 72
  Macros to Manipulate Device Numbers 72

4

Kernel Topics and Device Drivers

A first step in writing a device driver is deciding what sort of interface the device should provide to the system. The way in which `read()` and `write()` operations should occur, the kinds of control operations provided via `ioctl()`, and whether the device can be mapped into the user's address space using the `mmap()` system call, should be decided early in the process of designing the driver. (For simple memory devices that require neither DMA nor an `ioctl()` routine, and that don't interrupt, it's possible to use the `mmap()` system call to avoid writing a driver altogether. See the *Mapping Devices Without Device Drivers* section of this manual for more details).

Device drivers have access to the memory management and interrupt handling facilities of SunOS. The device driver is called each time the user program issues an `open()`, `read()`, `write()`, `mmap()`, `select()` or `ioctl()` system call, but only the *last* time the file is closed. The device driver can arrange for I/O to happen synchronously, or it can set things up so that I/O proceeds while the user process continues to run.

4.1. Overall Layout of a Character Device Driver

Here's a brief summary of the parts that comprise a typical device driver. In any given driver, some routines may be missing. In a complex driver, all of these routines may well be present. A typical device driver consists of a number of major sections, containing the routines introduced below.

Initial Declarations

Device drivers, like all C programs, begin with global declarations of various sorts. These declarations include the structures that the driver will overlay on the device registers. (These structures are often conveniently declared to contain unsigned integers and bit fields chosen to access various parts of the device registers). They also *must include* the declarations discussed in the *Autoconfiguration-Related Declarations* section of the *Overall Kernel Context* chapter of this manual.

Autoconfiguration Support

Then come the *xx*`probe()`, *xx*`attach()` and, perhaps, *xx*`slave()` routines. These are called at kernel boot time to determine if devices noted as being present in the config file are actually installed, and to initialize them if they are. This initialization may include the resetting of the interrupt vector.

Opening and Closing the Device

xx`open()` is called each time the device is opened at the user level; if multiple user processes open the device, *xx*`open()` is called multiple times. *xx*`close()`, in contrast, is called only when the *last* user process which is using the device closes it.

Reading to and Writing from the Device

xx`read()` and *xx*`write()` are called to get data from the device, or to send data to the device. Drivers for tty-like devices will probably structure *xx*`read()` and *xx*`write()` in the terminal-driver style (not described in this manual), while devices that deal simultaneously with groups of characters will probably have their *xx*`read()` and *xx*`write()` routines implemented in terms of a *xx*`strategy()` routine. Such *xx*`strategy()` routines are in every way subsets of block-driver *xx*`strategy()` routines — they are integrated with `physio()` and they use `buf` structures but they don't have anything to do with the kernel buffer cache. Character drivers for DMA device are likely to have `strategy()` routines, but they can be useful for non-DMA devices as well — as long as the devices do I/O in chunks.

Select Routine

xx`select` supports the `select()` system call, by which user processes can examine various devices (by way of I/O descriptors which specify them) to see if they are ready for reading, writing, or have an exceptional condition pending on them.

Start Routine

xx`start()` is needed in drivers that queue requests; it's called from *xx*`read()`, *xx*`write()` or *xx*`strategy()` to start the queue and is also called from *xx*`intr()` to send off the next request in the queue.

Mmap Routine

The `mmap()` routine is present in drivers for devices which are operated by being mapped into user memory — for example, frame buffers.

Interrupt Routines

There are two kinds of interrupt routines: polling (or auto-vectored) routines and vectored routines. Polling routines are necessary when the host system doesn't allow unambiguous means of mapping hardware interrupts to devices, as is the case with Multibus-based machines. Vectored-interrupt routines are used on VMEbus-based systems, which can map hardware interrupts immediately to devices. Drivers for VMEbus devices that are never run on Multibus-based systems need only vector interrupt routines, while drivers for devices which will be run on both Multibus and VMEbus machines need both types of interrupt routines. In this case the polling routine can determine the interrupting device and then call the vectored routine to do the rest.

Ioctl Routine

The *xx*`ioctl()` routine is called when the user process does an `ioctl` system call. These calls are the great escape hatches in the otherwise generally uniform I/O architecture. They are not, however, panaceas, and you should not overuse them to solve problems in driver design. Terminals have many `ioctl` calls, but they're a special case. They have many `ioctl` calls because they're inherently quite complex and yet SunOS still insists that they look like files.

4.2. User Space versus Kernel Space

SunOS, being a multi-tasking operating system, provides for multiple threads of control *at the user level.* (These multiple threads are the various user processes). At the kernel level, however, things are different. The SunOS kernel is *monolithic monitor* type of operating system, and, as such, it cannot be interrupted by user processes. Instead, it contains code which allocates its time (and other resources) among the various user processes, as well as to itself. *The kernel can be interrupted by hardware, but when handling interrupts it doesn't run on behalf of any individual user process.*

Device driver functions are invoked by kernel routines after user processes make system calls. These functions must be able to move data to or from user virtual space quickly and easily. Kernel functions are provided to help it do so, and to redundantly map memory so that it can be shared by user programs and the kernel.

Device drivers are parts of the kernel, and they inhabit kernel space:

- In the Sun-2 the kernel virtual address space is 16 megabytes, and is completely separate from the individual user virtual address spaces.
- In the Sun-3 and Sun-4, the kernel virtual address space is at the top of the current context, starting at `KERNELBASE`.
- In the Sun-4, the kernel uses the top 16 megabytes of the current Gigabyte context, starting at `0xFF000000`.
- In the Sun386i, the kernel uses the top 64 Megabytes. Of these, the kernel has 32 Mbytes reserved for its use; `kadb` has 16 Mbytes reserved, and the EPROM uses 16 Mbytes.

In general, drivers don't need to consider the details of kernel address-space implementation. Routines (like `copyin()` and `copyout()`) which deal in multiple address spaces will manage the details internally, as will programs like `kadb`.

4.3. User Context and Interrupt Context

A device driver can usefully be thought of as having a *top half* and a *bottom half.* The top half, consisting of the `read()`, `write()`, and `ioctl()` routines, and of any other routines which run on behalf of the user process making requests on the driver, is run at I/O request time. The routines in the top half make device requests that can cause long delays during which the system will schedule a new user process so that it can continue doing useful work. The bottom half, consisting of *xx*`intr()` and any routines that it may call, is run at hardware interrupt time.

Memory-mapped devices are usually not interrupt driven. Their drivers, thus, do not typically need to include interrupt routines. Memory-mapped devices operate by being written and read as system memory, and make no split-second demands (interrupt-time demands) upon their users.

After starting an I/O request, the top half calls `sleep()` to wait for the *bottom half* to indicate (by way of a call to `wakeup()`) that the request has been serviced. Thus, when a user program issues a read on (say) an A/D converter, it is normally suspended when the top half of the corresponding driver calls `sleep()` to wait until some input arrives. Alternatively, the top half of the driver can call `iowait()` and be put to sleep awaiting the completion of a buffer-oriented I/O transfer.

The top half contains not only all the non-interrupt time driver routines, but (for all practical purposes) the kernel routines above the driver as well. In particular, it contains the kernel `physio()` routine, which manages the decomposition of large I/O requests into a series of smaller ones that can be handled by the device.

The *bottom half* may include an *xx*`start()` routine, which can be called internally to start I/O. This allows the device-specific code to be isolated in one routine. *xx*`start()` is not a driver entry point. It's called from either *xx*`strategy()` or *xx*`intr()`, depending upon whether the device is busy or not.

Consider an A/D converter driver that expected the converter to interrupt when a sample was available. The kernel interrupt handler would detect the device interrupt and dispatch *xx*`intr()`, which would then store the sample data in a buffer and `wakeup()` the user process sleeping in the top half so it process could retrieve the data. If there was no user process sleeping in the top half, the `wakeup()` would have no effect, but the next process to read the A/D driver would find the data already there and wouldn't have to `sleep()`.

It must be stressed that, in general, *xx*`intr()` doesn't run on behalf of the current user process — this is, in fact, why it's distinguished so clearly from the top half. This means is that no information about the current user process is available, in any way, to *xx*`intr()`. It shouldn't examine, let alone change, any of the variables in the kernel `user` structure.

4.4. Device Interrupts

In general, the driver developer has limited control over the interrupt characteristics of the device. However, it should be said that some device-interrupt characteristics are better than others. In particular, interrupt-processing takes lots of time, and it's important that devices interrupt as seldom as possible. If, for example, a device can be made to handle multiple characters for each interrupt it processes, it should be. It's also preferable that a device not interrupt until its driver has enabled its interrupts, that it hold its interrupt line high until the driver asks that it be cleared, and that it remain quiescent after a bus reset (system boot).

Most hardware devices interrupt, and all interrupts occur at some given *priority level*. When an interrupt occurs, the system traps it, suspends the in-process operation (which may be a process entirely unrelated to the interrupting device or even the kernel) and resumes execution in the bottom half of the driver associated with the interrupting device. This means that the *top half* of a device driver can be interrupted *at any time* by its bottom half. If they wish to share data, they must do so in shared data structures, and they must take special provision to see that those structures remain consistent. An example of such a data structure is a pointer to a current buffer and a character counter. The top half of the driver must protect itself so that data structures can be updated as atomic actions, that is, the bottom half must not be allowed to interrupt during the time that the top half is updating some shared data structure. This protection is achieved by bracketing critical sections of code (sections that update or examine shared data structures) with subroutine calls that raise the processor priority to levels which can't be interrupted by the bottom half. Such a section of code looks like:

```
s = spln();
        . . .
         critical section of code that can't be interrupted
        . . .
(void)splx(s);
```

Here we've first raised the hardware priority level and then restored it after the protected section of code. (Determining the correct hardware priority will be discussed later). One section of code that almost always needs to be protected is the section where the top half checks to see if there is any data ready for it to read, or whether it can write data or start the device. Since the device can interrupt at any time, the section of code that checks for input in this fashion is wrong:

```
if (no input ready)
        sleep (awaiting input, software_priority)
```

because the device might well interrupt after the `if` condition is tested, but before the process switch. (The consequences, if this happens, are grave — the call to `wakeup()` will occur before the process has actually gone to sleep, and thus it will never wake up).

The above section of code must thus be rewritten to look like this:

```
s = spln();
while (no input ready)
        sleep (awaiting input, software_priority)
(void)splx(s);
```

If the top half executes the `sleep()` system call, the bottom half will be allowed to interrupt, because the hardware priority level is reset to 0 as soon as the `sleep()` context switches away from this process.

4.5. Interrupt Levels

In many cases it is possible to set the device interrupt level by setting switches on its board. If so, you must decide what processor-interrupt level the device is going to interrupt at. At first it may seem that your device is very high priority, but you must consider the consequences of locking out other devices:

- If you lock out the on-board UARTs (level 6) characters may be lost.
- If you lock out the clock (level 5) time will not be accurate, and the SunOS scheduler will be suspended.

- If you lock out the Ethernet (level 3), packets may be lost and retransmissions needed.
- If you lock out the disks (level 2), disk rotations may be missed.
- Level 1 is used for software interrupts and cannot be used for real devices.

In general, it's best to use the lowest level that will provide you with the response that you need.

4.6. Vectored Interrupts and Polling Interrupts

In Multibus-based Sun-2 machines, the kernel uses only auto-vectored (polling) interrupts. With auto-vectoring, the interrupt vector associated with a given device is based solely on the device interrupt priority level. Since many system configurations will contain more devices than there are interrupt levels, multiple devices may share the same interrupt level. Still, when processing an interrupt, the kernel must have a way of determining which device interrupted, and which driver should process the interrupt. In such configurations, the kernel proceeds by *polling* all the drivers at the given interrupt level (in the order that they are given in the config file), calling each of their polling interrupt routines in turn. These routines then proceed to interrogate their corresponding devices looking for the device that has an "attention bit" set, thus indicating that it issued the interrupt. Devices that don't indicate that they've interrupted can still be installed — one per system — by putting them at the end of the config file and thus at the end of the polling chain. Unclaimed interrupts can then be assumed to be from the last device.

After determining that one of its devices issued an interrupt, the polling routine services it and returns a non-zero to indicate that it did so (or a 0 to indicate that no device was found to originate the interrupt).

Polling only works if devices which share interrupt levels continue to interrupt until the driver tells them to stop. This is because the driver polling-interrupt routine returns to the kernel with an indication of which of the devices it has serviced. If two devices (A & B) are polling at the same interrupt level and *both issue an interrupt*, device A will always get serviced first. The kernel will then go on its merry way unless device B continues to interrupt. If it does, then when device A has been serviced, device B will be serviced. Fortunately, most Multibus boards continue to interrupt until told to stop. VMEbus boards typically do not, so it's important that they use vectored interrupts.

Sun VMEbus machines, (even those with Multibus devices installed by way of adapters) can take advantage of vectored interrupts. When handling a vectored interrupt, the kernel calls the appropriate driver's vector interrupt routine directly, passing it an argument to identify which of its devices (or controllers) interrupted.

It's important to realize that a driver can support both vectored interrupts *and* polling interrupts. Such a driver can be run on either type of machine, its polling interrupt routine will determine which device, if any, originated the interrupt, and then call the vectored interrupt routine to actually service it.

VMEbus devices — *if they interrupt* — are assigned unique identifying numbers in the range `0x40` to `0xFF` when they are described in the `config` file. It is these *vector numbers* that are used by the kernel to directly identify the interrupting device.

There are cases where no separate polling routine is needed. The first is where a driver *knows* that it supports only one device, and that no other device will share its device's interrupt level. In this case only an *xx*`intr()` routine need exist. It can then be specified in `mb_driver->mdr_intr` for use in the auto-vectored case *and* in the `config` file for the vectored interrupt case. Thus, all configurations will use the same interrupt routine. *Remember, this will only work if there are no other devices of any sort installed at the same interrupt level.*

The other case where *xx*`poll()` is not needed is when a driver will *never* support polling — presumably because it will never be run on a Multibus machine. In this case *xx*`intr()` should be specified in the `config` file for use as the vectored interrupt routine, and the auto-vectored (polling) interrupt routine specified in `mb_driver->mdr_intr` should be 0.

Note that in the first case above, where the device will have an interrupt level to itself, little need be done to make the driver work with vectored interrupts. One may simply take a polling interrupt routine, (perhaps renaming it *xx*`intr()` to avoid confusion) and install it as the vector interrupt routine by giving its name in the appropriate place in the `config` file. This isn't the most efficient thing to do, for when the routine is called through the kernel's vectoring mechanism, it will waste the information in its argument (which identifies the device originating the interrupt) and go on to poll its devices. Nevertheless it will work. It's better, however, if drivers contain both *xx*`intr()` and *xx*`poll()` routines, so that they may be easily transported to a variety of systems.

Another issue of concern only to drivers running on VMEbus machines is related to setting up the interrupt-vector number. When using the VMEbus-Multibus adapter or certain VMEbus devices, the vector number is set by switches on the circuit board. But some devices require that software initialize the device by telling it which vector number to use on interrupts. Presently, the only place where this can be done is in *xx*`attach()`. The vector number that *xx*`attach()` communicates to the device is in the `md_intr->v_vec` field of the `mb_device` structure — a `NULL` value in this field indicates that the host machine is Multibus based and does not support vectored interrupts.

A skeleton for a "typical" driver, one supporting both vectored and polling interrupts and using software to set interrupt vectors might look like:

```
/*
 * NXX is computed by config for each device type.
 * It can then be used within the driver source code to
 * declare arrays of device specific data structures.
 */

struct xx_device xxdevice[NXX];

/*
```

```
 * Attach routine for a device xx that must be notified of its
 * interrupt vector.
 */

xxattach(md)
    struct mb_device *md;
{
    register struct xx_device *xx = &xxdevice[md->md_unit];


#ifndef sun386
/*
 * Vector number given in kernel config file and passed by the autoconfiguration
 * process during boot. This code does not apply to the Sun386i, which does not
 * support vectored interrupts.
 */
    if (md->md_intr) {

        /* so we will be using vectored interrupts */

        /* WRITE interrupt number TO THE DEVICE */
        xx->c_addr->intvec = md->md_intr->v_vec;

        /* Setup argument to be passed to xxattach */
        *(md->md_intr->v_vptr) = (int)xx;

    } else { /* WRITE auto-vector code TO THE DEVICE */
        xx->c_addr->intvec = AUTOBASE + md->md_intpri;
    }
    /* any other attach code */
#endif
}

/*
 * Handle interrupt - called from xxpoll and for vectored interrupts.
 */
xxintr(xx)
    struct xx_device *xx;
{
    /* handle the interrupt here */
}

/*
 * Polling (auto-vectored) interrupt routine
 */
xxpoll()
{
    register struct xx_device *xx;
    int serviced = 0;

    /* loop through the device descriptor array */
    for (xx = xxdevice; xx < &xxdevice[NXX]; xx++) {
        if (!xx->c_present ||
            (xx->c_iobp->status & XX_INTR) == 0)
            continue;
        serviced = 1;
        xxintr(xx);
    }
    return (serviced);
}
```

4.7. Some Common Service Routines

The kernel provides numerous service routines that device drivers can take advantage of. The most important of these routines can be clustered into the functional groups given here. These routines, as well as many others, are described more completely in the *Kernel Support Routines* appendix to this manual:

Timeout Mechanisms

If a device needs to know about real-time intervals,

```
timeout(func, arg, interval)
    int (*func)();
    caddr_t arg;
    int interval;
```

is useful. `timeout()` arranges that after *interval* clock-ticks, the *func* is called with *arg* as argument, in the style *(*func)(arg)*. *interval* is often expressed as a multiple of the external variable `hz`, since `hz` gives the number of ticks per second on the host machine. (`10*hz`, then, specifies a timeout of ten seconds). Timeouts are used, for example, to provide real-time delays after function characters like new-line and tab in typewriter output, and to terminate an attempt to read a device if there is no response within a specified number of seconds. Also, the specified *func* is called at "software" interrupt priority from the lower half of the clock routine, so it should conform to the requirements of interrupt routines in general — you can't, for example, call `sleep()` from within *func*, although you can call `wakeup()`. (See also `untimeout()`).

Sleep and Wakeup Mechanism

Another key set of kernel routines is `sleep()` and `wakeup()`. The call

```
sleep(event, software_priority)
    caddr_t address;
    int priority;
```

makes the process wait (allowing other processes to run) until the *event* occurs; at that time, the process is marked ready-to-run. When the process resumes execution, it has the priority specified by *software_priority*.

The call

```
wakeup(event)
    caddr_t address;
```

indicates that the *event* has happened, that is, causes processes sleeping on the event to be awakened. The *event* is an arbitrary quantity agreed upon by the sleeper and the waker — it must uniquely identify the device. By convention, *event* is the address of some data area used by the driver (or by a specific minor device if there's more than one).

Processes sleeping on an event should not assume that the event has really happened when they are awakened, for `wakeup()` wakes *all* processes which are asleep waiting for the *event* to happen. Processes which are awakened should check that the conditions that caused them to go to sleep are no longer true.

Software priorities can range from 0 to 127; a higher numerical value indicates a less-favored scheduling condition. A distinction is made between processes sleeping at priority less than or equal to `PZERO` and those sleeping at numerically greater priorities.

If a process is blocked in `sleep()` at a priority less than or equal to `PZERO`, it will not be awakened upon receipt of a signal; the signal will not be processed until the process is awakened elsewhere and returns to user mode. (This means that a user cannot interrupt such a process by typing their interrupt character). Thus, it is a bad idea to sleep with priority less than or equal to `PZERO` on an event that may not occur.

On the other hand, if a process is blocked in `sleep()` at a priority greater than `PZERO`, and if a signal is sent to the process, it will be awakened. However, the call to `sleep()` will not return. This means that the routine that called `sleep()` cannot clean up after receiving the signal. If the routine needs to do such clean up, it can arrange for this by ORing the `PCATCH` flag into the priority it passes to `sleep()`. If this is done, and `sleep()` is interrupted by a signal, it will return 1; if the process is woken up normally, `sleep()` will return 0.

In general, sleeping at priorities less than or equal to `PZERO` should only be used to wait for events that occur quickly, such as disk and tape I/O completion. Waiting for events that may not occur quickly—for example, the typing of a particular key by a human at a keyboard—should be done at priorities greater than `PZERO`.

Incidentally, it is a gross error to call `sleep()` in a routine called at interrupt time, since the process that is running is almost certainly not the process that should go to sleep.

Raising and Lowering Processor Priorities

At certain places in a device driver it is necessary to raise the processor priority so that a section of critical code cannot be interrupted, for example, while adding or removing entries from a queue, or modifying a data structure common to both halves of a driver.

The `splx()` function changes the interrupt priority to a specified level, and then returns the old value.

For configuration reasons, the `pritospl()` macro is necessary to convert a Main Bus priority level to a processor priority level. The Main Bus priority level can be found in either `md->md_intpri` or `mc->mc_intpri`, where it is put by the autoconfiguration process. (These structures are defined in `/usr/include/sundev/mbvar.h`).

Here's how you normally use the `pritospl()` and `splx()` functions in a hypothetical `strategy()` routine:

```
hypo_strategy(bp)
    register struct buf *bp;
{
    register struct mb_ctlr *mc =
        hypoinfo[minor(bp->b_dev)];
    int s;

    s = splx(pritospl(mc->mc_intpri));
    while (bp->b_flags & B_BUSY)
        sleep((caddr_t)bp, PRIBIO);
    . . .
        here is some critical code section
    . . .
    (void) splx(s);        /* Set priority to what it was previously */
    . . .
}
```

Alternatively, `spln` can be used to set the processor to a certain fixed priority level.

Main Bus Resource Management Routines

On the Sun-2, Sun-3 and Sun-4, the routine `mbsetup()` is called when the device driver wants to start up a DMA transfer to the device, for DMA transfers require Main Bus resources. The `MBI_ADDR()` macro can then be used to transform the abstract integer returned by `mbsetup()` into a DVMA transfer address. At some later time, when the transfer is complete, the device driver calls the `mbrelse()` routine to inform the Main Bus resource manager that the transfer is complete and the resources are no longer required.

On the Sun386i, the `mbsetup()` and `dma_setup()` routines are called when the device driver wants to start up a DMA transfer. After the transfer is complete, the driver calls `mbrelse()` and `dma_done()`.

Data-Transfer Functions

The kernel provides a number of routines designed to transfer data between the user and kernel address spaces. These include `copyin()` and `copyout()`, general routines designed to move blocks of bytes back and forth. They also include `uiomove()`, `ureadc()` and `uwritec()`, routines which are designed to transfer data to or from a `uio` structure (see *Some Notes About the UIO Structure* in the *The "Skeleton" Character Device Driver* chapter for more details about this structure).

Kernel `printf()` Function

The kernel provides a `printf()` function analogous to the `printf()` function supplied by the C library for user programs. The kernel `printf()`, however, is more limited. It writes directly to the console, and it doesn't support `printf()`'s full set of formatting conversions. See the *Debugging with* **`printf()`** section of this manual for more details on the use of the kernel `printf()`.

Macros to Manipulate Device Numbers

A device number (in this system) is a 16-bit number `(typedef short dev_t)` divided into two parts called the *major* device number and the *minor* device number. There are macros provided for the purpose of isolating the major and minor numbers from the whole device number. The macro

```
major(dev)
```

returns the major portion of the device number *dev*, and the macro

```
minor(dev)
```

returns the minor portion of the device number. Finally, given a major and a minor number *x* and *y*, the macro

```
dev_t makedev(x,y)
```

returns a device number constructed from its two arguments.

5

Driver Development Topics

Driver Development Topics 75

- 5.1. Installing and Checking the Device 75
  - Setting the Memory Management Unit 75
    - Selecting a Virtual Address 76
    - Finding a Physical Address 79
    - Selecting a Virtual to Physical Mapping 79
      - Sun-2 Address Mapping 81
      - Sun-3 and Sun-4 Address Mapping 84
    - A Few Example PTE Calculations 87
  - Getting the Device Working and in a Known State 88
  - A Warning about Monitor Usage 90
- 5.2. Installation Options for Memory-Mapped Devices 90
  - Memory-Mapped Device Drivers 90
  - Mapping Devices Without Device Drivers 92
  - Direct Opening of Memory Devices 95
- 5.3. Debugging Techniques 97
  - Debugging with `printf()` 98
    - Event-Triggered Printing 100
  - Asynchronous Tracing 101
  - `kadb` — A Kernel Debugger 102
- 5.4. Device Driver Error Handling 103
    - Error Recovery 103
    - Error Returns 103

Error Signals 104

Error Logging 104

Kernel Panics 104

5.5. System Upgrades 105

5.6. Loadable Drivers 105

5

Driver Development Topics

5.1. Installing and Checking the Device

The central processor board (CPU) of the Sun Workstation has a set of PROMs containing a program generally known as the "Monitor". (See the appropriate *PROM Commands* chapter of the *PROM User's Manual* for detailed descriptions of the monitor commands and their syntax). The monitor has three basic purposes:

1) To bring the machine up from power on, or from a hard reset (monitor `k2` command).
2) To provide an interactive tool for examining and setting memory, device registers, page tables and segment tables.
3) To boot SunOS, stand-alone programs, or the kernel debugger `kadb`.

If you simply power up your computer and attempt to use its monitor to examine your device's registers, you will likely fail. This is because, while you may have correctly installed your device (a process that includes specifying its virtual memory mapping in the config file) those mappings are SunOS specific, and don't become active until SunOS is booted. The PROM will, upon power up, map in a set of essential system devices — like the keyboard — but your device is almost certainly not among them.

When installing a new device, you will use the monitor primarily as a means of examining and setting device registers. But before even beginning the development of your driver, it's a good idea to attach your device to the system bus and use the monitor to manually probe and test it. This will give you a chance to become familiar with the details of its operation, and to ensure that it works as you expect it to.

Setting the Memory Management Unit

Upon power-up, the PROM monitor:

- Maps the beginning of on-board memory, up to 6 megabytes, to low virtual addresses starting at virtual `0x0`.
- *Sun-2 machines only.* Maps the bus spaces into virtual address space, for the purpose of supporting Multibus devices. Multibus IO space is mapped from `0xEB0000` to `0xEBFFFF` on Sun-2 Multibus machines. On Sun-2 VMEbus machines, `vme16d16` is mapped from `0xEB0000` to `0xEBFFFF` so that Multibus cards attached by way of VMEbus adapter cards can be accessed. These two address spaces, Multibus I/O and `vme16d16`, are *not*

remapped by the SunOS kernel. This means that, for example, that kernel virtual address `0xEBEE40` can be used to talk to a device at `0xEE40` in Multibus IO space without setting up a mapping. (This shortcut is *only* possible for the two 16-bit Sun-2 spaces).

Later, using the autoconfiguration process, SunOS makes a pass through the config file (actually, through the `ioconf` file that was produced as output by `config` when it processed the config file). For each device, SunOS selects an unused virtual address (using an algorithm that doesn't presently concern us) and maps it into the device's physical address as specified in the config file.

SunOS then calls the *xx*`probe()` routine for each device, passing it the chosen virtual address. In this way, *xx*`probe()` is kept from having any knowledge of the physical address to which the device is mapped. *xx*`probe()` then determines whether or not the device is present. If it isn't, the virtual address can be reused.

To test a device, ignore the SunOS mappings and use the monitor to manually set the MMU to map your device registers to a known address in physical memory. Then you can use the monitor to verify its proper operation. This verification process will consist primarily of using the monitor's `O` (open a byte), `E` (open a word) and `L` (open a long word) commands to examine and modify the device's registers. Note that, in Sun-4 machines, words and long words are both 32 bits in length.

The process of setting up the device for initial testing consists of three discrete steps.

- The selection of an appropriate virtual address for the testing of the device.
- The determination of the physical address of the device, as well as the address space that it occupies.
- The use of the monitor to map the system's virtual address to the device's physical address. Detailed discussion of these three steps follow.

Since SunOS initializes the MMU in the course of its autoconfiguration process, it's possible to test a device by actually installing it, and then booting and halting SunOS. (You can halt SunOS by pressing the 'L1' and 'A' keys simultaneously, or, on a terminal console, by hitting the <BREAK> key). Having gotten to the monitor by this route, the MMU will be initialized to its SunOS run-time state. You can then use the monitor to test the device, or, if you wish, boot `kadb`. *(A hard reset — the monitor's* `k2` *command — will set the to MMU to its pre-SunOS power-up state). But while using the SunOS memory maps may occasionally be useful, it's not what you want to do during the first stages of device integration.*

Selecting a Virtual Address

First, understand that the MMU, when mapping a virtual address to a physical address, is actually mapping to a page of physical memory and an offset within that page. The low-order bits of a virtual address, those that specify the offset, *do not get mapped* — an address that is X bytes from the beginning of its virtual page will be X bytes from the beginning of whatever physical page it gets mapped into.

The mapping mechanism is the essentially the same for all Sun systems, although the details of address size and page mapping differ. This can be seen in the following diagrams:

Figure 5-1 *Sun-2 Address Mapping*

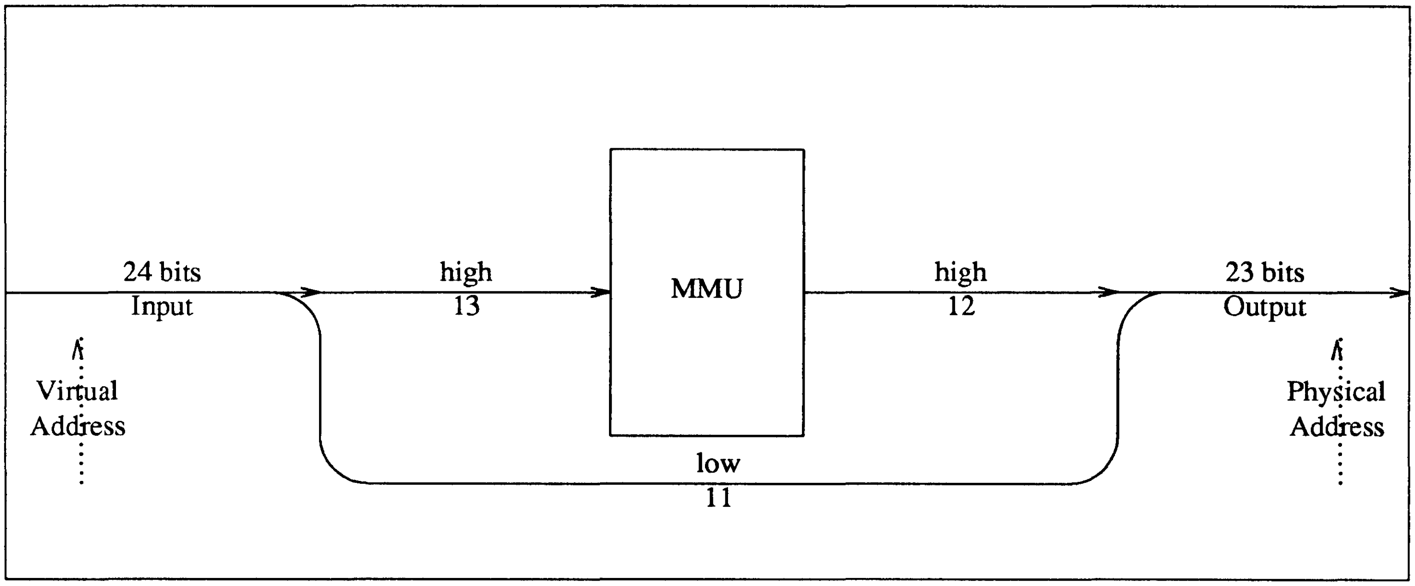

24 bits
Input
Virtual
Address
high
13
MMU
high
12
23 bits
Output
Physical
Address
low
11

Figure 5-2 *Sun-3 Address Mapping*

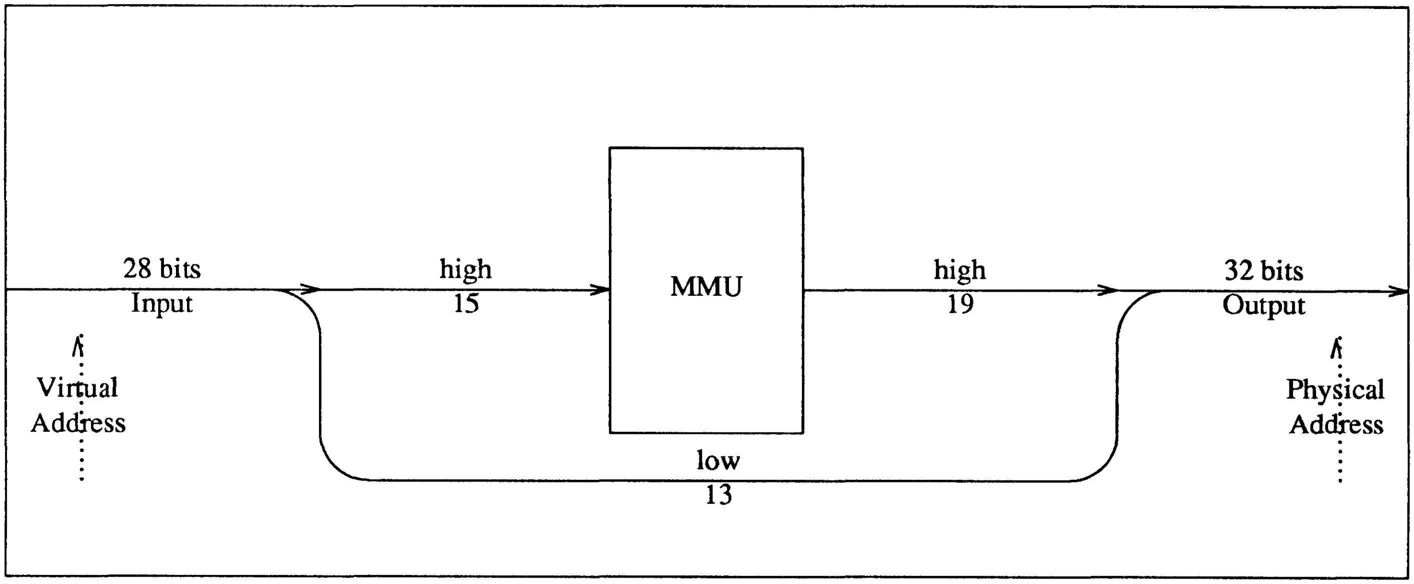

28 bits
Input
Virtual
Address
high
15
MMU
high
19
32 bits
Output
Physical
Address
low
13

Figure 5-3 *Sun-4 Address Mapping*

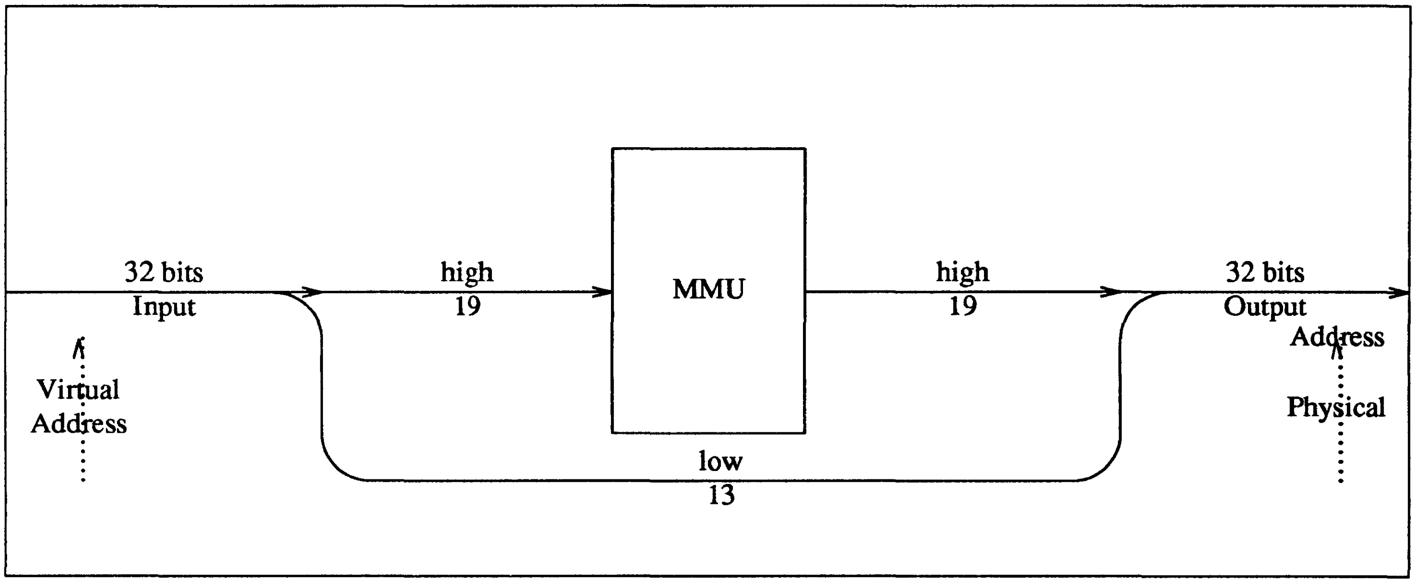
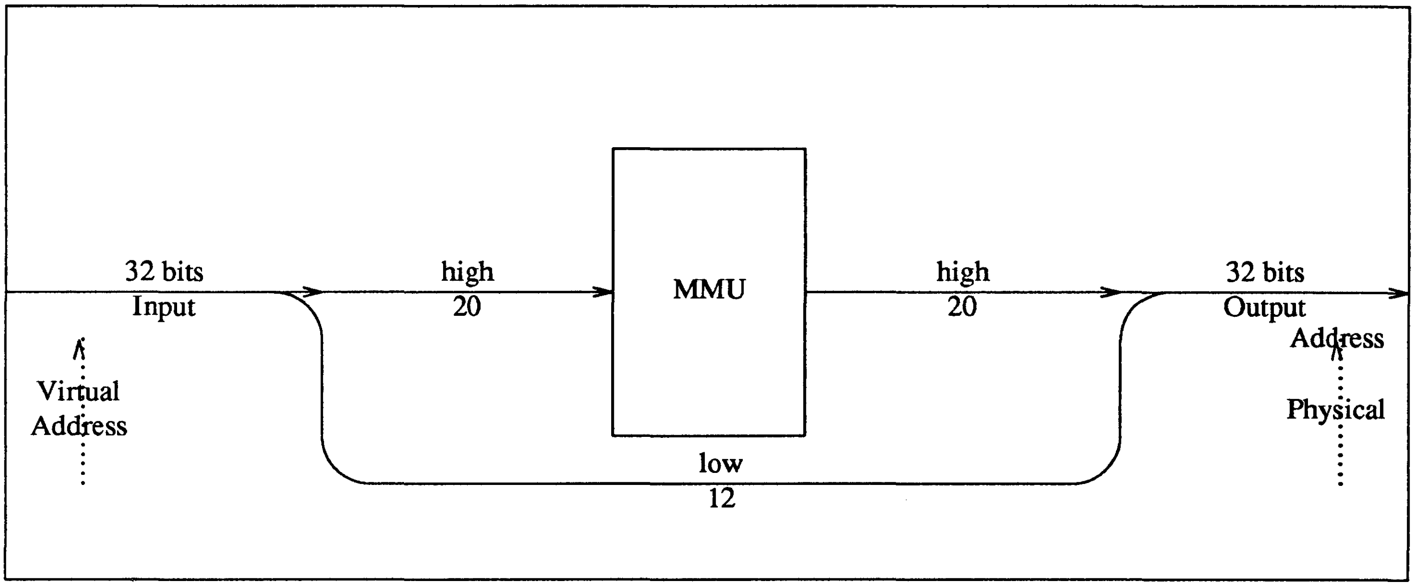

32 bits
Input
Virtual
Address
high
19
MMU
high
19
32 bits
Output
Address
Physical
low
13

Figure 5-4 *Sun386i Address Mapping*


32 bits
Input
Virtual
Address
high
20
MMU
high
20
32 bits
Output
Address
Physical
low
12

The easiest way to select a virtual address for PROM-monitor testing is to use one between `0x4000` and `0x100000` on Sun-2, Sun-3, and Sun-4 systems, or `0x20000` and `0x100000` on Sun386i systems. Addresses in these ranges are unused by the monitor in the respective Sun models, and are thus available.

(Note that these addresses, while convenient for testing, are not those that the kernel will choose when your device is finally installed).

It's most convenient to select a virtual address which has only zero's in its low-order bits. This way you select the first address in a virtual page. The low-order bits in the address you choose will remain unchanged. With `'X'` representing the unmapped low-order bits (11 for a Sun-2, 13 for a Sun-3 or Sun-4, 12 for a Sun386i the test address `0x4000` is, in binary:

```
    Sun-2:                0000 0000 0010 0XXX XXXX XXXX  (24 bits)
    Sun-3:           0000 0000 0000 100X XXXX XXXX XXXX  (28 bits)
    Sun-4:      0000 0000 0000 0000 100X XXXX XXXX XXXX  (32 bits)
  Sun386i:      0000 0000 0000 0000 0100 XXXX XXXX XXXX  (32 bits)
```

Finding a Physical Address

Your board may be preconfigured to some address. If it is, then use that address unless it conflicts with the address of an already installed device. If it does, you will have to find an unused physical address at which you can install your device. To do so, examine the kernel config file for the system upon which you are working. Tables in the *Hardware Context* chapter show memory layouts corresponding to typical configurations, but if your system has departed at all from the norm, you will have to consult your kernel's config file (to determine where devices have been installed) and the header files for the corresponding device drivers (to determine how much space they consume on the bus).

Selecting a Virtual to Physical Mapping

When selecting a virtual to physical mapping, it's best if you understand a bit about the internals of the Memory Management Unit. To this point we've only stressed that the MMU maps the top bits of the virtual address, leaving the offset bits unchanged. Now it will be necessary to explain the mapping process in more detail.

Some new concepts are necessary to discuss the details of virtual to physical memory mapping.

- The *context register (of real concern only on the Sun-2)* is a register specifying which of memory *contexts* should be used when mapping virtual addresses to physical addresses. Sun-2 and Sun-3 Context Registers contain 3 bits, and specify one of eight memory contexts; Sun-4/260 Context Registers contain four bits, and specify one of 16 memory contexts. Each SunOS *process segment* (containing either code, data or stack) is kept within a single memory context.
  - Sun-3s have user and kernel address spaces in the same hardware context. That is to say, there is only one virtual address space, a portion of which is used by the kernel and the rest by user processes. Sun-4 virtual address spaces are divided into two chunks. One of them is at the top of the addressable virtual memory space and the other is at the bottom. The size of the unused space between these two spaces varies with the model — in the Sun-4/260 each of the two virtual address spaces is 512 megabytes in size, and the space between them consumes 15 Gigabytes.

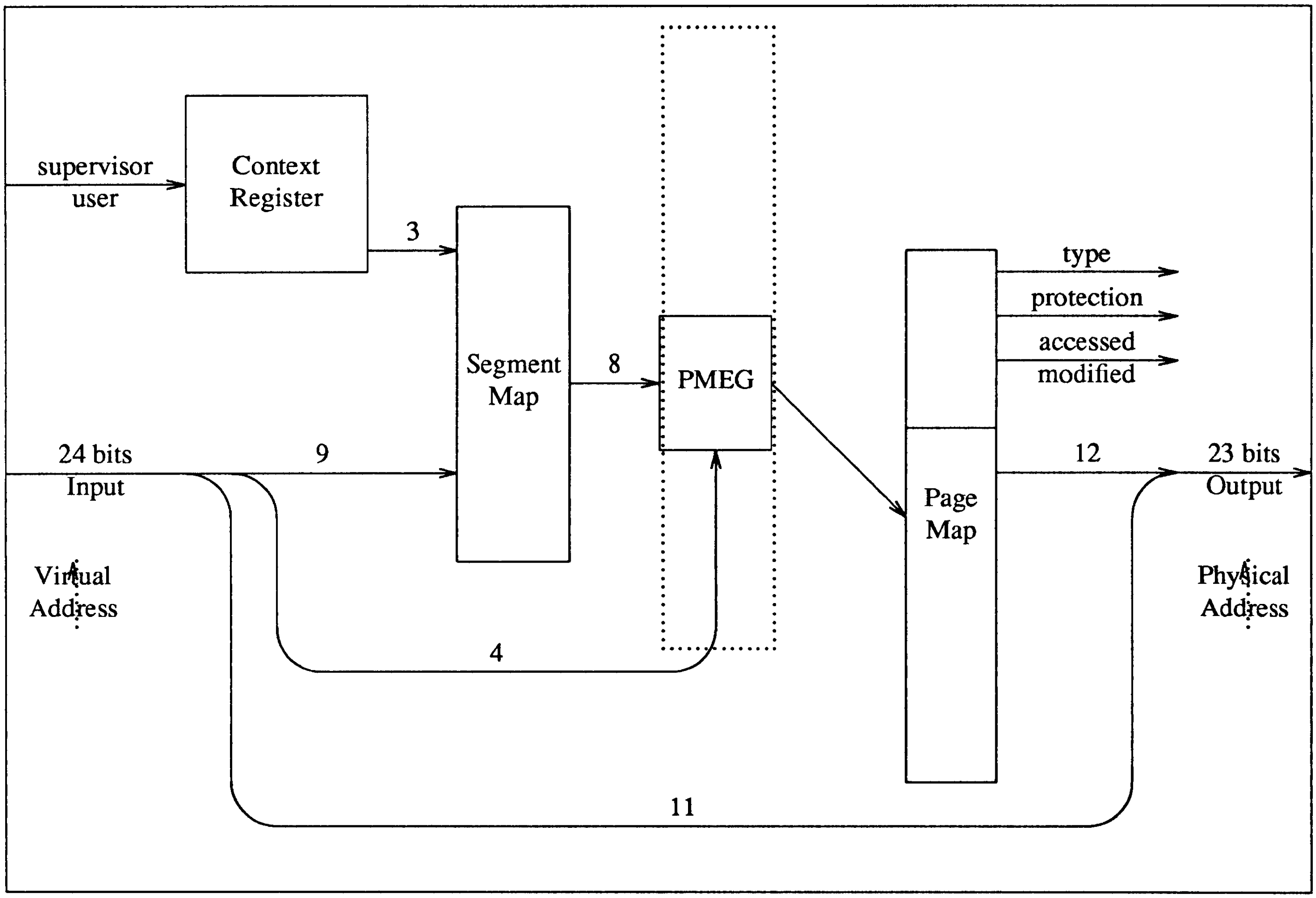
  - Sun-2s, on the other hand, segregate kernel and user processes into separate hardware contexts with separate address maps. Kernel processes are run in the supervisor context (context 0) and only processes in context 0 have access to the I/O devices.
- The *segment map* is used in conjunction with the *context register* to select the *page map entry group* (PMEG) corresponding to the virtual address being mapped. The eight bits in the segment register specify one of a group of 256 PMEGs.
- Within each *page map entry group* there are 16 *page table entries.*
- The *page map* maps the PMEG returned from the segment mapping with a second subfield of the incoming virtual address to exactly specify a single *page table entry* describing the physical page within which the virtual address is mapped.
- The *page table entry* (PTE) is the final output of the MMU. A PTE specifies the physical address of a page, as well as its type (e.g., on-board memory space), protection, and the state of its *access* and *modified* flags.

Note (for Sun-2 machines only): when testing your device, it's necessary to ensure both that you are in supervisor state and that you are in context zero (the kernel context). The monitor normally initializes to supervisor state, but if you enter it by way of an abort from SunOS, you will remain in whatever context you were in at the time of the abort. To be on the safe side, begin all of your monitor sessions with the command `S5`*. This will put you into supervisor data state, where you want to be. Note one important exception to this rule: if you've* `mmap()`*'ed the device into your (user) program's address space and want to check that this worked, you must use the* `S1` *command instead of the* `S5` *command. This will cause user function codes to be used when accessing page maps and data.*

Sun-2 Address Mapping

Note the following diagram of the Sun-2 MMU:

Figure 5-5 *Sun-2 MMU*


supervisor
user
Context Register
3
Segment Map
8
PMEG
type
protection
accessed
modified
24 bits
Input
9
12
23 bits
Output
Page Map
Virtual Address
Physical Address
4
11

Note that:

- The lower 11 bits of the incoming virtual address are passed through the MMU without being mapped — these are the bits that specify the position within the page, regardless of whether that page is physical or virtual.
- Multiple segment maps can specify the same PMEG, and often do.
- The PTE, on the output side of the MMU, specifies a variety of kinds of status information for the specified page, as well as the top bits of its physical address.

The process of mapping a virtual to a physical address consists, in practice, of plugging the right number into the right PTE. The monitor provides a simple means of addressing the right PTE, but you will have to calculate the right value to plug into it.

On Sun-2 systems, hardware PTEs are 32-bit numbers with the following structure.

V	r w x r w x	Type	a m	Unused (8)	Physical Page # (12)

Most of the PTEs that we will deal with will have similar structures, and so we can begin by making a "template" bit mask that we can use to construct our standard PTEs. One acceptable mask will assume values as follows:

```
V (valid) = 1
rwxrwx = 111111
(a/m) accessed/modified = 00
unused = 00000000
```

Thus, we can see that our template will be:

1	1 1 1 1 1 1	Type	1 1	0 0 0 0 0 0 0 0	Physical Page # (12)

This gives us a mask of `0xFE000000` (if we assume that the type field is `0000`). Now, as already mentioned, there are four types of memory, represented in the PTE by values of 0, 1, 2 and 3 in the type field indicated above. (Types 0 and 1 have the same meaning in both Multibus and VMEbus machines, but types 2 and 3 do not. Type 2 is used, on Sun-2 VMEbus machines, to designate the first 8 megabytes of the 24-bit VMEbus space — `0x0` to `0x7FFFFF` — and type 3 is used to designate the second 8 megabytes — `0x800000` to `0xFFFFFF`. (But remember that the top 64K of the 24-bit space is stolen for the 16-bit space). This use of two memory types to designate physical memory is necessary because the Sun-2 physical address size, 23 bits, is not sufficient to address all 16 megabytes of `vme24d16`.

Table 5-1 *Sun-2 PTE Masks*

Type	Description	Mask
0	On Board Memory	0xFE000000
1	On Board I/O Space	0xFE400000
2	(Multibus) Memory Space	0xFE800000
3	(Multibus) I/O Space	0xFEC00000
2	(VMEbus) VMEbus Low	0xFE800000
3	(VMEbus) VMEbus High	0xFEC00000

To determine the value which we need to plug into the PTE, we must add the appropriate mask to the appropriate physical page number, thus giving us the full 32-bit number that we need. Here, we will cease to explain details and simply give a series of rules for calculating physical page numbers.

```
If Sun-2 Multibus:

    If Multibus I/O Space, use Type-3 Template
    If Multibus Memory Space, use Type-2 Template

    Physical Page Number = Physical Address >> 11
```

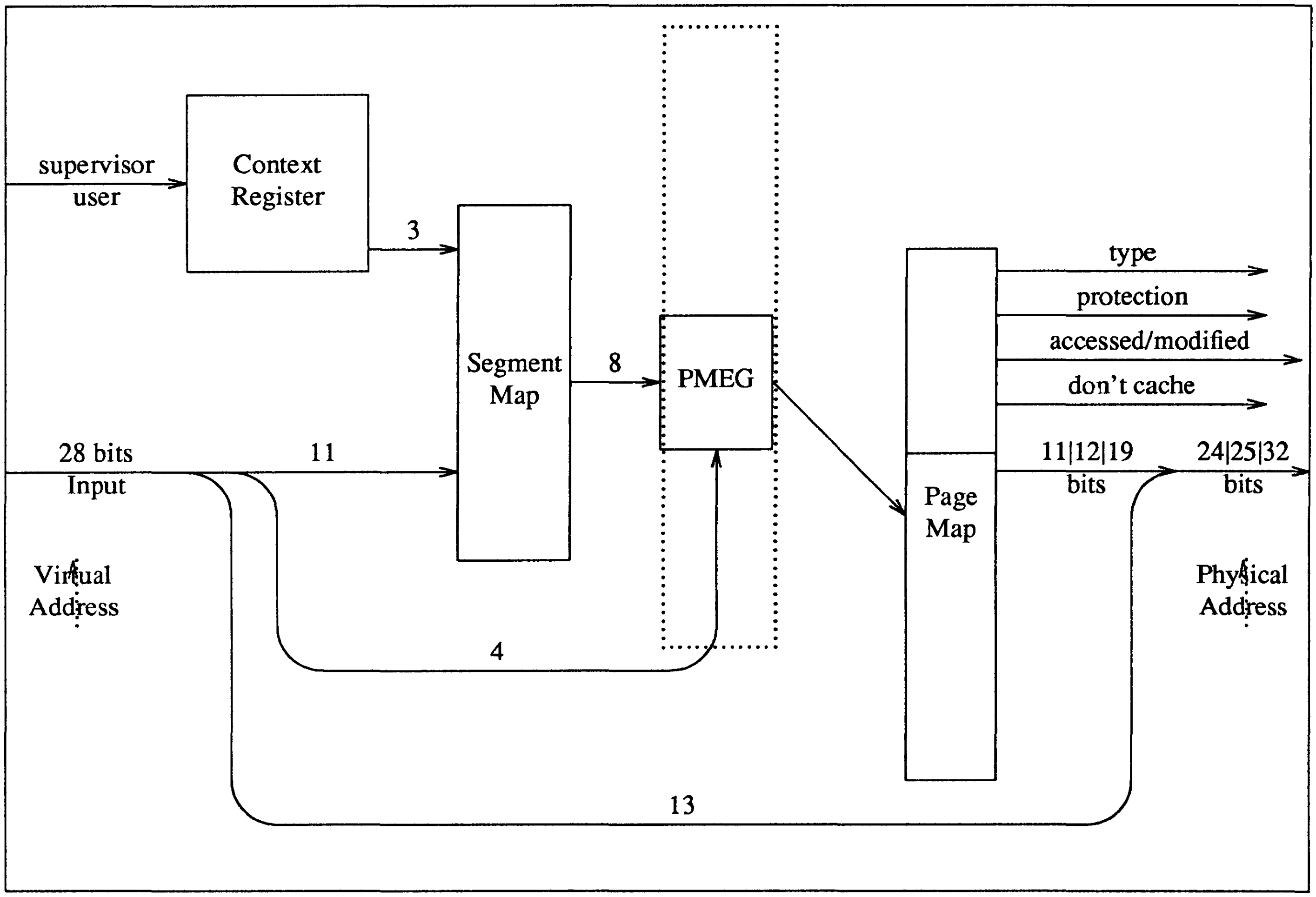
```
If Sun-2 vme24d16:

    If Physical Address >= 0x800000
        Use Type-3 Template
        Physical Page Number =
            (Physical Address - 0x800000) >> 11

    If Physical Address < 0x800000
        Use Type-2 Template
        Physical Page Number = Physical Address >> 11
```

```
If Sun-2 vme16d16

    Use Type-3 Template
    Physical Page Number =
        (Physical Address + 0x7F0000) >> 11
```

Sun-3 and Sun-4 Address Mapping

Consider the following diagram of address mapping on the Sun-3.

Figure 5-6 *Sun-3 MMU*

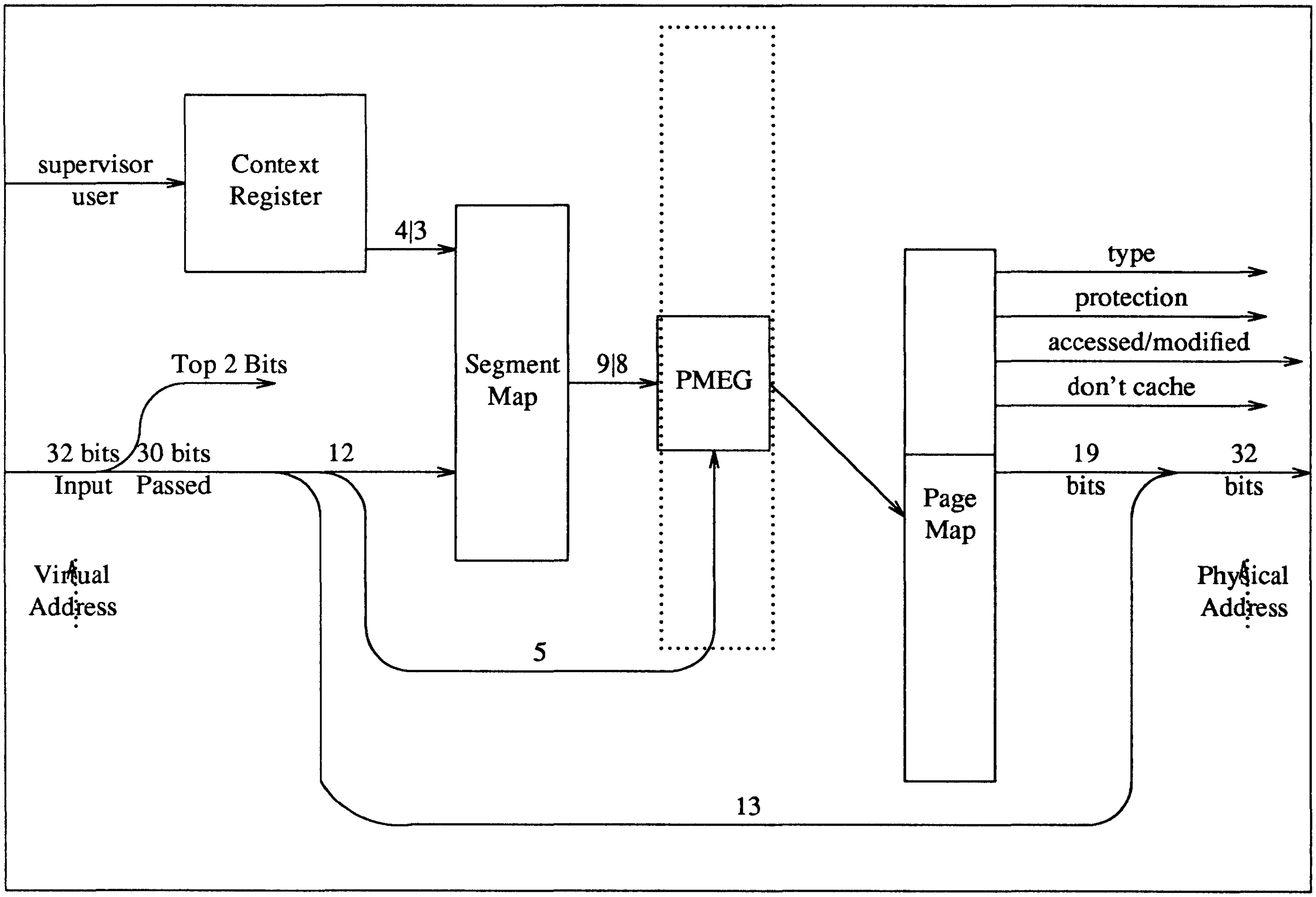

supervisor
user
Context Register
3
Segment Map
8
PMEG
type
protection
accessed/modified
don't cache
28 bits
Input
11
11|12|19 bits
24|25|32 bits
Page Map
Virtual Address
Physical Address
4
13

As you can see, the general scheme is the same as it was in the Sun-2, but the details have changed:

- The MMU is getting a 28-bit virtual address as its input, as opposed to a 24-bit address in the Sun-2.
- The number of mode and permission bits in the PTE has been reduced.
- The number of high-order bits reported out of the MMU, and thus the size of the physical address, is variable. The address size is fixed for any given Sun-3 machine, and varies only with the model — there are different kinds of Sun-3 machines and they have different physical address sizes.

The Sun-4 MMU is almost the same:

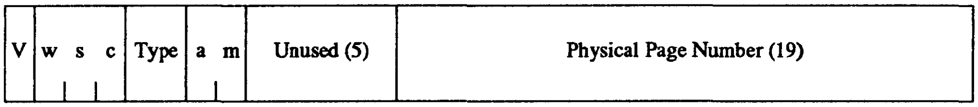
Figure 5-7 *Sun-4 MMU*


supervisor
user
Context Register
4|3
Segment Map
9|8
PMEG
Top 2 Bits
32 bits
30 bits
Input
Passed
12
Virtual Address
5
13
Page Map
type
protection
accessed/modified
don't cache
19 bits
32 bits
Physical Address

As you can see, the Sun-4 MMU is largely identical to the Sun-3 MMU. The differences are that:

- The Sun-4 MMU gets a 32-bit virtual address as its input, as opposed to a 28-bit address in the Sun-3. The top two bits are immediately shunted off. They must be either `00` or `11`, and are used to specify one of the two "chunks" in the virtual address space. (See *Selecting a Virtual to Physical Mapping* above).
- The number of bits coming off the Context Register is 4 (to specify one of 16 contexts) on Sun-4/260s and 3 (to specify one of 8 contexts) on Sun-4/110s.
- The number of bits coming off the Segment map is 9 for Sun-4/260s and 8 for Sun-4/110s.

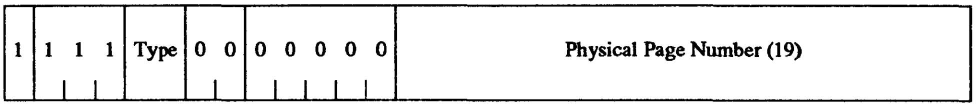
On both Sun-3 and Sun-4 systems, PTEs are 32-bit numbers with the following structure.



As we did with Sun-2 PTEs, we will make a "template" bit mask that we can use to construct our standard PTEs. One acceptable mask assumes values as follows:

```
V (valid) = 1
w/s (write ok/supervisor only) = 11
c (don't cache) = 1
unused = 00000
```

(A one (1) in the don't cache position only disables caching if the type is zero (0), since other types of pages are never cached). With the above values, our template will be:



This gives us a mask of `0xF0000000` (if we assume that the type field is `00`). Thus, the four masks for the four types of memory are:

Table 5-2 *Sun-3/Sun-4 PTE Masks*

Type	Description	Mask
0	On Board Memory	0xF0000000
1	On Board I/O Space	0xF4000000
2	`vme16d16`	0xF8000000
2	`vme24d16`	0xF8000000
2	`vme32d16`	0xF8000000
3	`vme16d32`	0xFC000000
3	`vme24d32`	0xFC000000
3	`vme32d32`	0xFC000000

To determine the value to be plugged into the PTE, we must add the appropriate mask to the appropriate physical page number, thus giving us the full 32-bit number that we need. Here, again, we will give rules instead of details.

```
If vme16d16
   or vme24d16
   or vme32d16

    Use Type-2 Template
```

1000 0000 0000 0000 0000 0000 => 400 | FC000000 FC000400

[illegible] 0000 0000 0100 0000 => 7FFF5 | FC000000 = FC07FFF6

10 000100
→ access type 3
& 800004

```
If vme16d32
   or vme24d32
   or vme32d32

    Use Type-3 Template
```

FC000000

```
If vme32d16
   or vme32d32

    Physical Page Number = Physical Address >> 13
```

400

```
If vme24d16
   or vme24d32

    Physical Page Number =
        (Physical Address +0xFF000000) >> 13
```

```
If vme16d16
   or vme16d32

    Physical Page Number =
        (Physical Address +0xFFFF0000) >> 13
```

A Few Example PTE Calculations

Example One: You wish to map a device which you have attached at physical `0x280008` onto bus type `vme24d16` on a Sun-3. You will map it at virtual `0xE000000`. What is the corresponding PTE?

> Well, since we are mapping the device into `vme24d16`, we will use `0xF8000000` as the template. Then, following the Sun-3 rules, as given above, we add the physical address to `0xFF000000`. This yields `0xFF280008`. In binary, this is:
>
> ```
> 1111 1111 0010 1000 0000 0000 0000 1000
> ```
>
> Shifting this right by 13 yields:
>
> ```
> XXXX XXXX XXXX X111 1111 1001 0100 0000
> ```
>
> Adding the template, `0xF8000000`, we get values for the 13 bits that are undefined from the shift. Thus the PTE is:
>
> ```
> 1111 1000 0000 0111 1111 1001 0100 0000
> ```
>
> Which is `0xF807F940`.

A final note: we've now calculated the PTE that maps the virtual page beginning at `0xE000000` to the physical page containing `0x280008`. To get the virtual address by which to access the device it's necessary to take the lower 13 bits of the physical installation address — the bits that are just passed through the MMU — and add them to virtual `0xE000000`. The lower 13 bits of physical `0x280008` are `0008`, and adding them to `0xE000000` yields `0xE000008`, the virtual address by which the device can be accessed.

Example Two: You wish to map physical `0xEE48` on bus type `vme16d32` on a Sun-3. Using virtual address `0xE000000`, what is the PTE?

> Since we are mapping the device into `vme16d32`, we will use `0xFC000000` as the template. Then, following the Sun-3 rules, as given above, we add the physical address to `0xFFFF0000`. This yields `0xFFFFEE48`. In binary, this is:
>
> ```
> 1111 1111 1111 1111 1110 1110 0100 1000
> ```
>
> Shifting this right by 13 yields:
>
> ```
> XXXX XXXX XXXX X111 1111 1111 1111 1111
> ```
>
> Adding the template, `0xFC000000`, we get values for the 13 bits that are undefined from the shift. Thus the PTE is:
>
> ```
> 1111 1100 0000 0111 1111 1111 1111 1111
> ```
>
> Which is `0xFC07FFFF`.

To get the virtual address by which to access the device at physical `0xEE48`, add its lower 13 bits, `0xE48`, to `0xE000000` — this yields `0xE000E48`.

Getting the Device Working and in a Known State

Before you even *think* about writing any code you should check out your device. You must get to know it, finding out early if it has any peculiarities that will affect its driver. It may, for example, have addressing and data-bandwidth limitations. Or, if it's a bus master, it may not implement the *release on request* bus-arbitration scheme the Sun supports. *Know the peculiarities of your device early, and then test it to verify that it's working before proceeding further with driver development.*

Make sure that the board is set up as specified in the vendor's manual. Device characteristics which, in general, have to be set properly before the device can successfully be used include:

- I/O register addresses for I/O mapped Multibus boards,
- Memory base addresses for Multibus boards that use Multibus memory space,
- Address and data widths,
- Interrupt levels,
- Interrupt vector numbers for VMEbus device,
- VMEbus address modifiers,

- The bus grant level for VMEbus devices should be set at 3.

Then, take down your system and power it off. Plug the device into the card cage and attempt to bring the system back up. If you can't boot the system, then there's a problem. Perhaps the board isn't really working, or perhaps it's responding to addresses used by other system devices. You must resolve this problem before proceeding further.

Take SunOS down again and attempt to contact the device using the PROM monitor. To do so, you will need to set up a PTE on the Sun-2, Sun-3, or Sun-4 which maps to the device's physical installation address. Use the procedures given above to calculate a PTE, then:

- Issue the monitor command that puts you into *supervisor data* state. This will be **s B** for Sun-4 machines and **s5** for all others. So, if you have a Sun-3, give the

  ```
  >s5
  ```

  command.
- Calculate, using the procedures given above, the PTE appropriate to the physical address you've chosen.
- Set the position in the kernel page map that corresponds to your physical address to contain the calculated PTE. This will map your chosen physical address, thus putting you in contact with your device. You may use the monitor's `P` command to perform this mapping. The `P` command takes a virtual address as its argument, displays the PTE that corresponds to that virtual address, and gives you the option of modifying the PTE. For example:

  ```
  >pF32000
  ```

  selects the page map entry that corresponds to the virtual address of `0xF32000` and displays it. It also displays a '?', which indicates that you may type in a new value to replace the one displayed. (See the appropriate *PROM Commands* chapter of the *PROM User's Manual* for more details). Note that all virtual addresses within a page select the same PTE.

Having contacted the device from the monitor, try some of the following:

- Try reading from the device status register(s), if there are any.
- Try writing to the device control and data registers(s), if there are any. Then try reading the data back to see if it got written properly (this assumes, of course, that the device allows the reading of these register(s).
- Try actually getting the device to do something by sending it data.
- If the device is a controller with separate slave devices, then switch a slave on and off and watch for changes in the controller status bits.

Your goal is to try to actually operate the device, for a moment, from the monitor. For example, if you have a line printer, try to print a line with a few characters. Be aware that bit and byte ordering issues are critical in this process. The reason you're doing this is to ensure that the device works and that you understand the way it works. When you understand the device's peculiarities, you can proceed to write a driver for it.

A Warning about Monitor Usage

When you use the monitor's `O`, `E` or `L` commands to open a location, the monitor *reads* the present contents of that location and displays them before giving you the option to rewrite them. In the best of all possible worlds, this would present no problems, *but many devices don't respond to reads and writes in as straightforward a fashion as does normal memory.*

For example, the Intel 8251A and the Signetics 2651 use the same externally addressable register to access *two* separate internal mode registers, and they have internal state logic that alternates accesses to the external register between the two internal registers. So suppose that you want to put something in mode register 1 of the 8251. You open the external register, the monitor displays its contents, and you then do your write. If, being cautious, you then read the external register to check that the data you wrote is there, you will find that it's not — because the read will sequence you on to the second register.

To deal correctly with such devices, it's necessary to use the monitor's "write without looking" facility and then read the locations back later to check them. You can write without looking with any of the monitor commands that "open" an area of memory; all that's necessary is that you enter a `value` after the `address` argument. For example:

```
>l [address] [value]
```

This will cause `value` to be written into `address` without first reading its current contents. For more information on hardware peculiarities and the problems that they can cause for the monitor, the *Hardware Peculiarities to Watch Out For* section of the *Hardware Context* chapter.

5.2. Installation Options for Memory-Mapped Devices

Memory-Mapped Device Drivers

Memory-mapped devices are the simplest types of devices to write drivers for. Frequently, however, their essential simplicity isn't obvious from a quick glance at their source code. This is because many memory-mapped devices are frame buffers, and frame-buffer drivers must set up and manage the low-level interface for the Sun window system as well as the standard device interface. Consequently, they tend to be littered with declarations and manipulations related to the "pixrect" (pixel rectangle) system. See the *Pixrect Reference Manual* for more details.

Memory-mapped devices are most frequently installed into Sun systems with simple drivers that map them into user address space (there are sometimes alternatives to such drivers, as you will see below). Such memory-mapped drivers don't really do much. Obviously, *xx*`probe()` and *xx*`mmap()` must exist, for the kernel must be able to check the device installation and perform the actual device mapping. And, in addition, *xx*`intr()` must be real if the device is interrupt driven. But *xx*`open()` and *xx*`close()` are usually stubs, and *xx*`read()` and *xx*`write()` can be calls to `nulldev`.

Keep in mind that the major purpose of a memory-mapped driver is to support the `mmap()` system call. This is very important because user processes which call window code must first map the frame buffer into their address space. They do so with the `mmap()` system call, which is translated by the kernel into a series of calls to the driver's `mmap` routine. Each of these calls returns page table entry information which the kernel needs to map a single page (the next page) of frame-buffer memory into a virtual address space. Here's some very simple driver *xx*`mmap()` code.

```
/*ARGSUSED*/
cgonemmap(dev,off,prot)
    dev_t dev;
    off_t off;
    int prot;
{
    return (fbmmap(dev,off,prot,NCGONE,cgoneinfo,CG1SIZE));
}

/*ARGSUSED*/
int fbmmap(dev, off, prot, numdevs, mb_devs, size)
    dev_t dev;
    off_t off;
    int prot, numdevs;
    struct mb_device **mb_devs;
    int size;
{
    int kpfnum;

    if ((u_int) off >= size)
        return -1;

    kpfnum =
      hat_getkpfnum(mb_devs[minor(dev)]->md_addr + off);
    return kpfnum;
}
```

dev is, of course, the device major and minor number, and *off* is the offset into the frame buffer (passed down from the user's `mmap()` system call). *prot* is also passed down from the user's call, but it is not currently used. As you can see, there's a bit of shuffling around and then a call to `hat_getkpfnum`, which returns a Page Frame Number which *xx*`mmap()` is expected to return.

Note that `mb_dev->md_addr` is the address of the frame buffer from the Main Bus device structure. This is the device installation address as given in the kernel config file. The offset is checked to be sure the user isn't mapping beyond the end of the frame buffer.

Mapping Devices Without Device Drivers

Under a restricted set of circumstances, it's possible to avoid writing a device driver altogether by using the `mmap()` system call to overlay the device's registers and memory onto user memory. Having done this, you can read and write the registers — as if they were normal user memory — from a user program.

What this really amounts to is piggybacking the new device onto an another, system standard, virtual memory device (and its driver). The `mmap()` routine of a system virtual memory device is then used to do the user-device mapping, and the "installation" is accomplished without the development of a driver specific to the user device. Instead, a user level program is written, one that calls the `mmap()` system call.

The restrictions on this shortcut are, however, fairly severe.

- The device must not require any special handling of the type that would go into `xxioctl()`.
- The device (including all its control registers) must work with user function codes, since that's what it will get when mapped into and then accessed from user space.

NOTE *MC680X0 processors, SPARC processors and the Intel 80386 all run in either 'user' or 'supervisor' state. Many devices, in turn, restrict certain of their operations, and will only perform them when the processor is in supervisor state. The Sun CPU is in supervisor state only when executing kernel code. This means that device drivers, which are part of the kernel, can issue device commands which are not available from user processes. Also note that, when the CPU is in supervisor state, as it is when driver code is executing, the device will receive different VMEbus address modifier codes than when the CPU is in user state. For details about these codes see the VMEbus specification).*

- The device must not require any other sort of special handling — it cannot, for example, be multiplexed, interrupt driven, or do DMA.
- Finally, there are security problems associated with this sort of installation. Since the system virtual-memory devices are normally owned by and restricted to the superuser, your programs will either have to change their permissions to allow normal users to access them, or will have to run with superuser privileges. The former strategy is usually not acceptable in the long run, because it creates a gaping hole in the security of the system. And it's far from clear that the second alternative is desirable either.

The virtual-memory devices of interest here are those that support mapping over the entire range of a virtual address space. They are:

Table 5-3 *Virtual Memory Devices*

Machine Type	Memory Device Name
Multibus (Sun-2 only)	`mbmem`
Multibus (Sun-2 only)	`mbio`
VMEbus	`vme16d16`
VMEbus	`vme24d16`
VMEbus (Sun-3 and Sun-4)	`vme32d16`
VMEbus (Sun-3 and Sun-4)	`vme16d32`
VMEbus (Sun-3 and Sun-4)	`vme24d32`
VMEbus (Sun-3 and Sun-4)	`vme32d32`
ATbus (Sun386i only)	`atmem`

In addition, there are memory pseudo-devices that support access to the on-board devices that users are allowed to access. These are `/dev/fb`, `/dev/mem` and `/dev/kmem` (See the `mem(4)` manual page for details).

`/dev/fb` is a memory device which, on any given system, is set up to address the local frame-buffer device. It can be used as if it were a system memory device — on any given system, `/dev/fb` can be `mmap()`'ed into user memory and then written to, with the effect of writing the local frame buffer memory.

To use `mmap()` with one of the system memory devices, you must do three things:

- Open the device.
- Calculate the *offset* which you will need to call `mmap()`. This offset is merely the device address on the appropriate system memory device rounded to a page boundary. That is to say that you get the offset from the device manual and/or the switches on the device itself.
- Call `mmap()` to allocate virtual space and map in the physical bus address of your device, which you must know. (See the *Hardware Context* chapter for a discussion on how to pick a good physical address from the information in the system config file).

The following example program uses `/dev/fb` rather than one of the virtual memory devices. It makes a good example because it maps the system frame buffer into user memory so that it can then be written from a user program. It uses `mmap()` to set things up, but doesn't bother with calling `munmap()`, because unmapping occurs automatically when the memory device is closed. This close occurs implicitly when the program ceases execution. (The machine segment size is 128K for the Sun-2 and Sun-3; 256K for the Sun-4; and 4Mbytes for the Sun386i. Areas greater than the machine segment size should be mapped only with special care. For details, see the discussion of `mmap()` in the *User Support Routines* appendix).

Once the device has been mapped into user space it can be treated as a piece of local user memory. (Remember that memory accesses performed by way of this mechanism will be reflected — at the device level — as non-privileged (user) accesses. This is because `mmap()` accesses inherit the privilege of the process that calls `mmap()`. Thus, if memory is mapped by a driver, subsequent accesses to it will have the standard supervisor data access privilege, but if it's called from a user process, as described here, subsequent accesses will be non-privileged. Attempts to access supervisor-only device registers without supervisor privilege might produce a bus error, i.e., they're inaccessible from a user program, and thus a kernel level driver must be written to manipulate them. The device will also receive different address modifier codes when accessed from a user process than when accessed via a device driver).

```
#include <stdio.h>
#include <sys/file.h>
#include <sys/mman.h>
#include <sys/types.h>

/* Width and Height of Frame Buffer in Bits */
#define WIDTH 1152
#define HEIGHT 900

main()
{
    int fd;
    unsigned len;
    char *addr;

    /* Open the frame-buffer device */
    if ((fd = open("/dev/fb",O_RDWR)) < 0)
         syserr("open");

    /* Compute total number of bytes */
    len = ((WIDTH * HEIGHT)/8);

    /*
     * offset  must be page aligned.   /dev/fb
     * is already aligned with frame-buffer memory
     */
    offset = 0;

    /* Map device memory to user space */
    addr = mmap((caddr_t)0, len, PROT_READ|PROT_WRITE,
        MAP_SHARED, fd, 0);
    if (addr == (caddr_t)-1)
        syserr("mmap failed");

    writeFB(addr);
    exit(0);
}

writeFB(addr) /* Write to frame buffer */
    char *addr;
{
    char color;
    int i,j;

    color = 0xFF;
    for (i = 0; i < HEIGHT; i++) {
        color = ~color;
        for (j = 0; j < WIDTH/8; j++)
            *addr++ = color;
    }
}

syserr(msg) /* print system call error message and terminate */
    char *msg;
{
    extern int errno, sys_nerr;
    extern char *sys_errlist[];

    fprintf(stderr,"ERROR: %s (%d", msg, errno);
    if (errno > 0 && errno < sys_nerr)
        fprintf(stderr, "; %s)\n", sys_errlist[errno]);
    else
        fprintf(stderr,")\n");
    exit(1);
}
```

NOTE *This example uses the special memory device* `/dev/fb`, *since this device is always set up to address the frame buffer memory.*

So, despite the plethora of limitations on the sorts of devices that can be installed by way of mapping them into user space, it's quite an easy thing to do. If your device characteristics are such that this is an option, you may well wish to take it. And even if such an installation isn't an attractive long-term option (for example, because of unacceptable security problems) it may still be attractive as a short-term alternative to driver development. Even in environments where security considerations make it unacceptable in the long term, it can allow you to get your device up and running very quickly. Sometimes this counts for a lot.

Direct Opening of Memory Devices

It should be noted, for the purpose of completeness, that there's another approach to avoiding driver development, one that's even easier than the use of `mmap()` described here, and even more limited. That is, it's possible to simply open the virtual memory device that contains your board, to seek to the location of its registers, and then to read and write those registers as if they were regular memory.

This approach has most of the same problems as does the use of `mmap()`, *and is notable mainly because, with it, the device receives supervisor function codes. It does, however, introduce new problems.* It doesn't give you the same degree of control as does mmap(), and you often need that control when dealing with devices. When you use mmap(), the device actually becomes part of your user memory space, and it's left to the compiler to generate exactly the I/O accesses which you implicitly specify in your structure and variable declarations. You can always access exactly what you want, and the accesses occur directly as *move byte* and *move word* operations. Thus they are very fast.

When, however, you simply open a system memory device as a file and then read and write to it, your communication with your board is mediated by the I/O system. The I/O systems will always try to do the "right thing" (if you request I/O at an odd address or for an odd number of bytes it will perform byte access as appropriate; otherwise it will use short integers), but it still doesn't give you the kind of control that can be had using `mmap()`. Furthermore, I/O operations involve lots of code, and take ***hundreds*** of times as long as direct references to `mmap()` 'ed references, which proceed by way of the MMU and use low-level store and move instructions to directly access device registers and memory as physical memory.

So the bottom line is that, unless you need to access a device only a few times, or if you need to receive supervisor function codes (and the corresponding VMEbus address-modifier codes) and performance isn't critical, you can do your installation by opening a system memory device and then seeking to your device registers and memory space. Otherwise, use `mmap()` or write a driver. If you do decide to use the `open()/lseek()` method, do so with low-level I/O rather than with the standard I/O library. The standard I/O library implements a buffered I/O scheme which will add considerably to your problems.

The following user program is similar to the example above, in that it writes the same pattern to the memory of a frame buffer. This time, though, the write is done by way of the I/O system rather than by using `mmap()`, and the frame buffer is taken to be installed at *OFFSET* (whatever the device physical installation address is) in the `vme24d16` memory space.

NOTE *Since all Sun VMEbus machines have a built-in, on-board frame buffer, this example is only meaningful for color frame buffers. On Sun-2 Multibus machines, however, this code would work with* `/dev/obmem` *and an offset of* `BW2MB_FB`.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/param.h>
#include <sys/buf.h>
#include <sys/file.h>

void syserr();
long lseek();

/* Width and Height of Frame Buffer in Bits */
#define WIDTH 1152
#define HEIGHT 900

main()
{
    int fd;
    /* Open the system memory device containing the frame buffer */
    if ((fd = open("/dev/vme24",O_RDWR)) < 0)
         syserr("open");

    /* Seek to the frame buffer memory */
    if (lseek(fd, (long)OFFSET, L_SET) == -1L)
        syserr("lseek");

    writeFB(addr);
    exit(0);
}

writeFB(fd) /* Write to frame buffer */
    int fd;
{
    char color;
    int i,j;

    color = 0xFF;
    for (i = 0; i < HEIGHT; i++) {
        color = ~color;
        for (j = 0; j < WIDTH/8; j++) {
            if (write(fd, &color, 1) == -1)
                syserr("write");
        }
    }
}
```

5.3. Debugging Techniques

As described above, it's a good idea to begin debugging by using the monitor to check that the device has been installed at the intended address, and that it works, before proceeding to debug your device driver. This allows you to avoid debugging the device simultaneously with the driver, and experience that you'd like to avoid for as long as possible. Alternatively, if you're confident in both your device and the correctness of your installation, you can simply make a new kernel, boot it and proceed with debugging. In this case you should put some `printf()` messages — see below — into the *xx*`probe()` routine. Then you can at least see the device get contacted and initialized.

Debugging drivers is significantly more difficult than debugging regular user programs, for a number of reasons:

- In the first place, device drivers are part of the system kernel. This means that the system is not protected from their errors. Addressing errors, for example, will frequently trip hardware traps and crash the system.
- As mentioned above, there's the possibility that the device hardware will be buggy. For this reason, you can't really trust your environment in the same way as you can when writing a user program on a mature computer system.
- Some device behave in rather peculiar ways. (See *A Warning about Monitor Usage*, above).

- Finally, the debugging environment in the kernel is thinner than it is in user space. There is a kernel debugger, `kadb`, and this a a big step towards making life easier for driver developers. Still, life remains more difficult when debugging in kernel space.

  It's possible to prototype drivers in user address space by using techniques similar to those described in the Mapping Devices Without Device Drivers section of this chapter. The same constraints given there apply to prototyping. In particular, it's not possible to run an interrupt routine, or to probe for non-existent devices without generating bus errors from prototype drivers in user space. If the device generates no interrupts, and if it doesn't do DMA, the entire driver might be able to run in user space.

For all these reasons, you should give extra care to desk-checking your code, and check a reference manual when not absolutely sure of the meaning of a given construction. Don't take chances.

Also, make changes incrementally. Don't try to save time by making many changes at once. You will save time in the long run if you take the time to add and test a few parts at a time. Keep your feet on solid ground.

Use trace output from `printf()`, as described below. Drivers can act in surprising ways, and the best way to proceed is by making the flow of operations highly visible.

NOTE *On the Sun386i system, the loadable drivers feature makes driver development much easier because the code-compile-reboot-test cycle is reduced to code-compile-load-test.*

Debugging with `printf()`

With the availability of `kadb`, the kernel debugger, the importance of `printf()` in the debugging of device drivers has been significantly reduced. Still, even with `kadb` available, `printf()` statements remain useful as means of providing synchronous tracing of overall driver flow and structure. `kadb` can be made to provide a similar sort of tracing (by tying print commands to strategically chosen breakpoints) but this won't altogether eliminate the `printf()` statement. The `printf()` has long found application in driver debugging, and, as a matter of taste and experience, some programmers will continue to use it. For this reason, we will discuss its use in some detail.

The kernel `printf()` sends its message directly to the systems console, without going through the tty driver. As a consequence, the printing is uninterruptible—the characters aren't buffered. Furthermore, `printf()` runs at high priority, and no other kernel or user process activity takes place while its output is being produced. `printf()` thus radically limits overall system performance (though this is usually ok while device drivers are being debugged).

The window systems should not be up when you use `printf()` to debug a driver because its output will go to the console window. On the Sun386i system, it is best to set the global variable `newlog` to 0.

There is a second kernel print statement, `uprintf()`. `uprintf()`, however, is of little use to driver developers. It attempts to print to the current user tty as identified in the `user` structure, and prints to the console only if there's no current user tty (at which it becomes identical to `printf()`). `uprintf()` cannot be called from lower-half routines, which run in interrupt context and cannot make any assumptions about the `user` structure (where `uprintf()` looks to determine the current user tty). `uprintf()` is most useful for production drivers, like tape drivers that encounter media errors, which want to report errors not to a programmer but to the user.

There are occasions in which the use of `printf()` *(or* `uprintf()`*) statements will change the behavior of your driver.* `printf()` *statements, for example, can affect the timing of operations in the driver being tested as well as in other drivers. The output may be so slow relative to other device operations that interrupts are lost and system failures are introduced; thus, it is frequently impossible to synchronously trace a device interrupt routine. Driver code may begin to fail only when* `printf()`*s are introduced, or, even worse, only when* `printf()`*s are disabled. If you' re debugging a tty driver, you may even face a situation where* `printf()`*-based tracing generates new calls to the driver being debugged. Thus, there are situations in which it cannot be used. In such situations, you should use* `kadb` *or the techniques suggested below in the section on Asynchronous Tracing.*

The best way to use `printf()` statements for tracing driver execution is by setting things up so that you can toggle printing by using the kernel debugger, `kadb` (see below) to set and reset print-control variables. Doing so is very simple. At the top of the driver source file, include statements like:

```
#ifdef XXDEBUG
int xxdebug = 0;
#define XXDPRINT if (xxdebug > 0) printf
#endif
```

(It's important that the variables like `xxdebug` be global, so that you can later access them freely from the debugger — remember that all drivers are part of one program, the kernel, and name your print-control variables so as to avoid naming conflicts).

Then, instead of calling `printf()` inside the driver routines, call `XXDPRINT`. Each call should be in the form:

```
#ifdef XXDEBUG
XXDPRINT("driver name...",...);
#endif
```

which will only call `printf()` if `XXDEBUG` is defined and `xxdebug` is set to a value greater than 0.

Make sure that each call to `XXDPRINT` identifies the driver, for it's possible that you, or some other programmer, will want to see debugging output from several drivers at once. And leave the debugging code in for a while after you're finished — bugs may surface later.

Having set things up like this, you can turn the `printf()`'s on or off at any time by using `kadb` to set unset or change the print-control variable `xxdebug`. Or you can use `adb` if you wish, running it at user level in a separate window:

```
example adb -w /vmunix /dev/kmem
```

(`adb` won't allow you to set breakpoints in the kernel, but it will allow you to set and unset variables — you can change the value of `xxdebug`, or even reset a variable which has caused your driver to hang). *Remember that you're in the kernel and BE CAREFUL.*

Incidentally, `/dev/kmem` represents the kernel *virtual* address space, which is why it's used here. `adb -k /vmunix /dev/mem`, in contrast, generates a view of the *physical* address space, because `/dev/mem` represents the physical memory. This latter command is useful for examining core files.

Good places to put `printf()` statements include:

- driver routine entry points
- before critical subroutine calls
- upon reading status information from the device
- before writing of commands or data to the device
- at intermediate points in complex routines
- at routine exit points

Note again that you don't have to restrict yourself to a single `xxdebug` variable, or to binary tests that check to see if a variable is on or off. You can use as many variables, and as many values for each variable, as necessary to reflect the functional divisions most appropriate to your driver. It might even be useful to get truly esoteric, and send certain trace statements directly to the user tty (by calling `uprintf()`) while the rest use `printf()` and go to the console.

Event-Triggered Printing

In the above discussion, the `xxdebug` variable was initialized by the compiler, and toggled with a debugger. However, it's just as easy to have the driver routines themselves set a trigger variable under pre-chosen conditions.

For example, if you wanted to enable tracing after a given *condition* had occurred, you could declare `xxdebug`, just as was shown above, but define `XXDPRINT` somewhat differently:

```
#ifdef XXDEBUG
int xxdebug = 0;
#define XXDPRINT(v,msg,a1,a2) \
    if (xxdebug > (v)) printf(msg,a1,a2);
#endif
```

and then, in the code that checks for the condition:

```
#ifdef XXDEBUG
if (condition) xxdebug = 1;
#endif
```

Then to call `XXDPRINT`:

```
#ifdef XXDEBUG
XXDPRINT(0,"driver name...\n",a,b);
#endif
```

One major disadvantage of using the kernel `printf()` is that its output doesn't go through a device driver, and thus can't be paused with Control-S or redirected to a file. It's possible, then, that `printf()` will overwhelm you with output. There are a number of things that you can do if you run into this problem:

- If you haven't used multivalued print-control variables, then do so. This gives you more control than you have with simple on/off print control, and will allow you to reduce the amount to trace noise.
- You can use a debugger to set the global variable `noprintf`. This will keep `printf()`'s output from being sent to the console, but that output will still go to a buffer where kernel error messages are kept before being transferred to `/var/adm/messages`. You can examine the message buffer at your leisure, in one of two different ways:
    - From a user window, you can use `dmesg`.
    - From `kadb` (or `adb` on `/dev/kmem`) you can type `msgbuf+10/s`.
- It's also possible to reconfigure your system so that it uses a hardcopy terminal as its console over a RS-232 line. Then, you won't lose any of the `printf()` output.
- Best of all, you can get another machine and connect it to your machine over a RS-232 line. Having done so, use `tip` to open a window on the second machine *as the console of the test machine*. You can then use `tip`'s record feature (see the `tip` man page) to make a record of all the stuff that `printf()` is sending to the test machine's console.

Asynchronous Tracing

As mentioned above, there are occasions when timing problems forbid the use of the printf statement. In these cases, it's a good idea to give up any attachment that you might have to `printf()` statements and use `kadb`.

Or, if you prefer, it's possible to deal with timing problems by using `kadb` to patch the `noprintf` variable, and then to check the message buffer to see what's going on. Doing so:

- allows you to continue using the debugging code that you installed before encountering the timing problem, and
- presents you with a sequential list of the events in your driver, a list spelled out in English phrases and including interrupt-level events.

Or, you can simply use `kadb` for everything.

`kadb` — A Kernel Debugger

NOTE `kadb` *does not work with versions of the kernel earlier than 3.2.*

`kadb` is an interactive debugger similar in operation to `adb`. `kadb` differs in several key respects from `adb`. It runs as a standalone program under the PROM monitor, rather than as a user process in user address space. And it allows you to set breakpoints and single step in the kernel!

Thus, running a kernel under `kadb` is significantly different than running it under `adb -k`. The `k` option to `adb` merely makes it simulate the kernel memory mappings while `kadb` actually runs in the kernel address space. And unlike `adb`, which runs at user level and as a separate process from the process being debugged, `kadb` runs in system space as a *coprocess*. It shares not only the kernel address space but its CPU supervisor mode as well.

`kadb`, for all intents and purposes, is a version of `adb`. It has the same command syntax and almost the same command set. Thus, you can see the `kadb` and `adb` manual pages, as well as *Debugging Tools for the Sun Workstation*, for more details on its use. Note, however, the following points of special interest to driver developers:

- All interrupts are disabled while interacting with `kadb` (except non-maskable interrupts). Thus, when using `kadb` to examine memory, the kernel remains stable. However, while single stepped instructions are being executed, the actual standing priority of the kernel is temporarily restored, and interrupts can get dispatched, run and return. You won't notice unless you have a break point set in the interrupt routine, which works just fine.
- `kadb` is installed so that, when a program is being run under it, an abort sequence (L1-A) will transfer control not to the PROM monitor but to `kadb` itself. Once in `kadb`, you can abort again and be transferred to the monitor. The transfer is direct and immediate, so you can use the monitor to examine control spaces (e.g. page and segment maps) which are not accessible from `kadb`. The monitor `c` command will return you to `kadb`.
- `kadb` runs in the same virtual memory space as the kernel itself, and with the CPU in supervisor mode. This means that `kadb` uses the kernel memory maps when calculating virtual addresses, and that it can directly examine kernel structures. This is in contrast to the situation with `adb -k`, where software copies of the page table entries are used to map virtual addresses to physical pages.
- `kadb`'s memory view is almost the same as that resulting from `adb /vmunix /dev/kmem`. In other ways, however, `kadb` is much different. To give just one example: on Sun-3 machines, where users and supervisors share the virtual address space, `kadb` allows the user to examine the user virtual address space (this is *not* true with `adb -k`).
- Finally, be aware that `kadb` — as a consequence of the way that `adb` works — always does 32-bit memory reads. Even if you tell `kadb` to read a byte it will read a long. This leads to a lack of control that can cause problems when reading device registers. (This problem does not exist on the Sun386i.

On the Sun386i, when `kadb` is told to read a byte, it does. Within `kadb`, the `B` command is used to read a single byte and the `v` command to write one).

5.4. Device Driver Error Handling

There are various types of errors: "expected" errors like those generated by *xx*`probe()` routines, transient errors in operations that can reasonably be retried, fatal errors that require controlled shutdowns, and others. The kinds of errors that you will face depends upon the kinds of drivers that you write and the peculiarities of your devices; few generalizations can usefully be made.

To further complicate matters, the detection and treatment of errors varies greatly from device to device. You should begin by carefully reading your device specification manual to determine the error indications that can arise and the responses that should be made when they do. Most devices have at least an error bit in the control/status register, and usually more detailed error information is available. Ideally, you should understand all potential errors, avoid those that you can and recover from the rest. This ideal isn't always achievable, but try not to leave any obvious holes. *If you do nothing else, check for device errors and use the kernel* `printf()` *function to report them to the system console.*

There are various error reporting and management mechanisms available to the driver developer. Most of them have already been mentioned as they've become relevant; here they are collected and summarized:

Error Recovery

It's difficult to generalize about error-recovery mechanisms, for they are largely device specific. It's worth noting, however, that:

- Some errors are worth retrying and some aren't; the matter is entirely device specific.
- Error-recovery routines should be able to run at the interrupt level. This is because errors can occur either synchronously or asynchronously with respect to the dispatch of device commands, and, therefore, recovery routines must be callable from interrupt routines.
- If you do implement error recovery logic, you must do so consistently. The data structure that contains retry-status information must be global, and must be protected by critical sections. Error-recovery routines, like interrupt routines in general, must take special pains to protect data-structure integrity; indeed, they must *restore* such integrity upon encountering errors they can't recover from.

Error Returns

There are three mechanisms by which driver routines can report errors up to their calling routines. The first, of course, is by way of the values that the driver routines return to their callers. The second, and most important, is the error-reporting mechanism based upon the buffer-header. *This is the only mechanism that can be used when returning errors from* *xx*`strategy()`, *xx*`start()`, and *xx*`intr()`. (See the discussion of *xx*`intr()` error reporting in the *Summary of Device Driver Routines* chapter. Finally, it is possible to directly set the global error register, `u.u_error`, from routines in the top half of the driver.

Error Signals

It is sometimes desirable to have a driver send a software interrupt to the process or processes. It's possible, for example, that a device will fail in an unrecoverable fashion — in this case it's perhaps a good idea to signal the user processes, rather than merely returning an extraordinary error code. It's also possible (though rare) for a driver to encounter serious errors from which it can recover by restarting the device — user processes may also need to be notified in this case. The kernel `psignal()` and `gsignal()` routines can signal either a single process or all the processes in a given process group.

Error Logging

When you use the kernel `printf()` statement to report errors to the console, those errors are also placed into a system error-message buffer accessible to the `dmesg` daemon. `dmesg` can be, and typically is, run every 30 minutes by the `crontab` daemon, for the purpose of appending the messages in the buffer to `/var/adm/messages`. Note that the message buffer is small, and that if a lot of entries are being written into it, some of them will get lost before being transferred into `/var/adm/messages`.

Kernel Panics

The most unequivocal way of dealing with an error is to panic when you get it. The `panic()` routine is provided to help you do so in a somewhat controlled fashion — `panic()` is called only on unresolvable fatal errors. It prints "panic: mesg" on the console, and then reboots. (Or, if you're running under the debugger, it transfers control to `kadb`). `panic()` also keeps track of whether it's already been called, and avoids attempts to sync the disks (by flushing all pending write buffers) if it has, since this can lead to recursive panics.

The final production version of a driver should call `panic()` only when "impossible" situations are encountered; lesser errors should be recovered from. During debugging, though, `panic()` can be used to implement a passable assert mechanism.

```
#ifdef XXDEBUG
if (inconsistent condition)
    panic("Assertion Failed: ...");
#endif
```

(It's possible to write a fancier assert mechanism, for example by having an ASSERT macro which calls an `assert()` routine which prints error context information and then calls `panic()`, but this minimal hack will perhaps do).

Finally, note that in cases where it's *very* important to halt the system *immediately* after detecting an inconsistent condition, `kadb` can be used. The driver code can test for the inconsistent condition, and then set a debugging variable:

```
if (inconsistent condition)
    junk = 1;
```

`kadb` can then be used to set a breakpoint at the machine instruction generated from the assignment to `junk`.

5.5. System Upgrades

System upgrades generally have minimal effects on user-written device drivers. The changes that are necessary are rare and release specific.

Some changes must be made if user-written drivers are to work with new release software. In Release 2.0, for example, there was a minor change in one of the bus-interface structures. There wasn't much involved in adapting user-written drivers, but it had to be done.

In other cases, changes are optional. When VMEbus machines were introduced, for example, drivers had to be adapted to run on them; however, it was possible to upgrade Multibus machines without rewriting user-written drivers.

In any case, any release upgrades that imply changes — either optional or mandatory — to user-written device drivers will be documented in the *System Summary and Change Notes* for the release in question.

5.6. Loadable Drivers

The Sun386i supports loadable drivers. This feature allows you to add a device driver to a running system without rebooting the system or rebuilding the kernel. The loadable drivers feature reduces time spent on driver development, and makes it easier for users to install drivers from other vendors.

This section explains how to convert a non-loadable driver to be a loadable driver.

Conversion of a non-loadable driver to a loadable driver requires an initialization or "wrapper" module to be written. The module `zzinit.c` below is an example of a wrapper module that contains the same kind of information ordinarily provided by a config file and by the linker. Almost all wrappers are identical to the example below. Usually, only the actual structure initialization values are different.

The following module is a typical example of an initialization routine for a driver named `zz` that has one controller and one device on that controller.

```
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/buf.h>
#include <sys/param.h>
#include <sys/errno.h>
#include <sundev/mbvar.h>
#include <sun/autoconf.h>
#include <sun/vddrv.h>

extern zzopen(), nulldev(), zzstrategy(), zzdump();
extern zzsize(), zzread(), zzwrite(), zzioctl();
extern zzint(), nodev(), seltrue();

extern struct mb_driver zzcdriver;    /* defined in driver */
```

```
/*
 * Driver block device entry points (normally in <sun/conf.c>)
 */
struct bdevsw zzbdev = {
    zzopen, nulldev, zzstrategy, zzdump, zzsize, 0
};

/*
 * Driver character device entry points (normally in <sun/conf.c>)
 */
struct cdevsw zzcdev = {
    zzopen, nulldev, zzread, zzwrite, zzioctl, nodev,
    nulldev, seltrue, 0
};

/*
 * Controller structure (normally in ioconf.c) (see <sundev/mbvar.h>)
 */
struct mb_ctlr zzcctlr[] = {
    &zzcdriver, 0, 0, (caddr_t) 0x00001000, 2, 6,
    SP_ATMEM, 0
};

/*
 * Device structure (normally in ioconf.c) (see <sundev/mbvar.h>)
 */
struct mb_device zzcdevice[] = {
    &zzcdriver, 0, 0, 0, (caddr_t) 0x00000000, 0, 0, 0x0,
    0, 0x0
};

/*
 * The following structure is defined in <sun/vddrv.h>
 *
 * If the number of controllers is 0, then the address of the
 * controller structure array must be NULL. Similarly, if the number
 * of devices is 0, then the address of the device structure array
 * must be NULL. The bdevsw or cdevsw entries can be NULL if there
 * is no block or character device for the driver.
 */
struct vdldrv vd = {
    VDMAGIC_DRV,      /* Type of module. This one is a driver. */
    "zzdrv",          /* Name of the module. */
    zzcctlr,          /* Address of the mb_ctlr structure array */
    &zzcdriver,       /* Address of the mb_driver structure */
    zzcdevice,        /* Address of the mb_device structure array */
    1,                /* Number of controllers */
    1,                /* Number of devices */
    &zzbdev,          /* Address of the bdevsw entry */
    &zzcdev,          /* Address of the cdevsw entry */
    0,                /* Block device number. 0 means let system choose. */
    0,                /* Char. device number. 0 means let system choose.*/
};
```

```
/*
* This is the driver entry point routine. The name of the default entry point
* is xxxinit. It can be changed by using the "-entry" command to modload.
*
* inputs: function code - VDLOAD,  VDUNLOAD, or VDSTAT.
*       pointer to kernel vddrv structure for this module.
*       pointer to appropriate vdioctl structure for this function.
*       pointer to vdstat structure (for VDSTAT only)
*
* return: 0 for success. VDLOAD function must set vdp->vdd_vdtab.
*       non-zero error code (from errno.h) if error.
*
*/

xxxinit(function_code, vdp, vdi, vds)
    unsigned int function_code;
    struct vddrv *vdp;
    addr_t vdi;
    struct vdstat *vds;
{
    switch (function_code) {
    case VDLOAD:
            vdp->vdd_vdtab = (struct vdlinkage *)&vd;
            return (0);
        case VDUNLOAD:
            return (unload(vdp, vdi));
        case VDSTAT:
            return (0);
        default:
            return (EIO);
    }
}

static unload(vdp, vdi)
        struct vddrv *vdp;
        struct vdioctl_unload *vdi;
{
    extern struct buf zztab;

    struct buf *dp;

    dp = &zztab;
    if (dp->b_actf) {
        return(-1); /* The driver still has an active request. */
    }

    /* The driver can do any device shutdown stuff that it needs to do */

    return(0);
}
```

Your driver routines can be placed in the wrapper module if you like. If your driver is big, it is more appropriate to break it into several modules.

If you decide to place your driver in the wrapper module, then the driver can be compiled with the following command line:

```
example# cc -c -O -Dsun386 -Di386 -DTTYSOFTCAR -DWEITEK \
    -DVDDRV -DCRYPT -DVPIX -DIPCSHMEM -DIPCSEMAPHORE \
    -DIPCMESSAGE -DLOFS -DNFSSERVER -DNFSCLIENT -DUFS \
    -DINET -DSUN386 -DKERNEL -Umc68000 -Di386bug zzinit.c
```

However, if the driver consists of more than one module, then you must use the link editor, `ld`(1), with the `-r` option to preserve relocation information. For example you might type:

```
example# cc -c -O -Dsun386 -Di386 -DTTYSOFTCAR -DWEITEK \
    -DVDDRV -DCRYPT -DVPIX -DIPCSHMEM -DIPCSEMAPHORE \
    -DIPCMESSAGE -DLOFS -DNFSSERVER -DNFSCLIENT -DUFS \
    -DINET -DSUN386 -DKERNEL -Umc68000 -Di386bug zzinit.c

example# cc -c -O -Dsun386 -Di386 -DTTYSOFTCAR -DWEITEK \
    -DVDDRV -DCRYPT -DVPIX -DIPCSHMEM -DIPCSEMAPHORE \
    -DIPCMESSAGE -DLOFS -DNFSSERVER -DNFSCLIENT -DUFS \
    -DINET -DSUN386 -DKERNEL -Umc68000 -Di386bug zz1.c

example# cc -c -O -Dsun386 -Di386 -DTTYSOFTCAR -DWEITEK \
    -DVDDRV -DCRYPT -DVPIX -DIPCSHMEM -DIPCSEMAPHORE \
    -DIPCMESSAGE -DLOFS -DNFSSERVER -DNFSCLIENT -DUFS \
    -DINET -DSUN386 -DKERNEL -Umc68000 -Di386bug zz2.c

example# ld -r -o zz.o zzinit.o zz1.o zz2.o
```

Thus the object module can be created either by the `cc`(1) command, when the driver resides in one module, or by the `ld`(1) command, when the driver resides in several modules.

In either case the resulting object file (`zzinit.o` or `zz.o`) is a normal COFF file and can then be used with the `modload` command.[5] The driver is just like any other program, except its text segment starts somewhere in the range `0xFD000000` to `0xFE000000`.

---

[5] "COFF" = Common Object File Format, a UNIX object-file standard to which Sun386i assembler and link-editor output files (normally `a.out`) comply. See `coff`(5).

6

The ‘‘Skeleton’’ Character Device Driver

The ‘‘Skeleton’’ Character Device Driver **111**

6.1. General Declarations in Driver 114
6.2. Autoconfiguration Procedures 115
  `probe()` Routine 115
  `attach()` Routine 117
6.3. **`open()`** and **`close()`** Routines 117
6.4. **`read()`** and **`write()`** Routines 119
  Some Notes About the UIO Structure 120
6.5. Skeleton **`strategy()`** Routine 121
6.6. Skeleton **`start()`** Routine 122
6.7. **`intr()`** and **`poll()`** Routines 124
6.8. **`ioctl()`** Routine 126
6.9. Skeleton Driver Variations 126
  DMA Variations 126
    Multibus or VMEbus DVMA 126
    A DMA Skeleton Driver 127
  Variation with ‘‘Asynchronous I/O’’ Support 130
    Select Routines 131
    Adding Asynchronous Notification 134
    Adding an **`ioctl()`** routine 134

6

The ''Skeleton'' Character Device Driver

This chapter presents one of the simplest drivers you could ever hope to encounter, a driver for an imaginary Multibus character device known as the "Skeleton" device. Both programmed I/O and DMA versions of the driver will be discussed. There is a complete version of this driver in the *Sample Driver Listings* appendix to this manual — the parts are presented piecemeal here with some discussion of their functions.

What we're doing here is inventing the very simple, I/O mapped, Skeleton controller. It's actually a "free device" with no separate controller and no separate slaves. It has a single-byte command/status register, and a single-byte data register. It's a write-only device. It's not a slow tty-type device — you can provide vast blocks of data and the Skeleton board gets it all out very fast. It interrupts when it's ready for a data transfer, and comes up in the power-on state with interrupts disabled and everything else in neutral.

Note: the Skeleton device is capable, in both its simple and its DMA variants, of writing chunks (not to say "blocks") of data in a single operation. It is, therefore, a character device that can make good use of *xx*`strategy()` routines, `physio()`, `buf` structures and other block-I/O mechanisms. As explained in *Kernel Topics and Device Drivers,* its use of these mechanisms does *not* make it a block driver. Rather, its simple needs are a subset of the needs of block drivers, and it's convenient here for form to follow function.

Let us assume that we've installed the Skeleton board with its control/status register at `0x600` in Multibus I/O space — this puts its data register at `0x601`. The control/status register is both a read and a write register, with bit assignments as shown in the tables below.

BIT	7	6	5	4	3	2	1	0
Read	Interrupt				Device Ready	Interface Ready	Error	Interrupt Enabled

BIT	7	6	5	4	3	2	1	0
Write						Reset		Enable Interrupt

Here is a brief description of what the bits mean:

When *reading* from the status register

bit 7 is a 1 when the board is interrupting, 0 otherwise.

bit 3 is a 1 when the device that the board controls is ready for data transfers.

bit 2 is a 1 when the Skeleton board itself is ready for data transfers.

bit 0 is a 1 when interrupts are enabled, 0 when interrupts are disabled.

When *writing* to the status register

bit 2 resets the Skeleton board to its startup state — interrupts are disabled and the board should indicate that it is ready for data transfers.

bit 0 enables interrupts by writing a 1 to this bit, disables interrupts by writing a 0.

The header file for this interface is in `skreg.h`. By convention, we put the register and control information for a given device (say `xy`) in a file called `xyreg.h`. The actual C code for the `xy` driver would by convention be placed in a file called `xy.c`. The header file for the Skeleton board looks like this:

```
/*
 * Registers for Skeleton Multibus I/O Interface -- note the byte swap
 */
struct sk_reg {
	char sk_data;	/* 01: Data Register */
	char sk_csr;	/* 00: command(w) and status(r) */
};

/* sk_csr bits (read) */
#define SK_INTR        0x80    /* Device is Interrupting */
#define SK_DEVREADY    0x08    /* Device is Ready */
#define SK_INTREADY    0x04    /* Interface is Ready */
#define SK_ERROR       0x02    /* Device Error */
#define SK_INTENAB     0x01    /* Interrupts are Enabled */

#define SK_ISTHERE     0x0C    /* Existence Check;
              Device and Interface Ready */

/* sk_csr bits (write) */
#define SK_RESET       0x04    /* Reset Device and Interface */
#define SK_ENABLE      0x01    /* Enable Interrupts */
```

The complete device driver for the Skeleton board consists of the following parts:

`skprobe`
: is the autoconfiguration routine called at system startup time to determine if the `sk` board is actually in the system, and to notify the kernel of its memory requirements.

`skopen` and `skclose`
: routines for opening the device for each time the file corresponding to that device is opened, and for closing down after the last time the file has been closed.

`skwrite`
: routine that is called to send data to the device.

`skstrategy`
: routine that is called from `skwrite()` via `physio()` to control the actual transfer of data.

`skstart`
: routine that is called for every byte to be transferred.

`skpoll`
: the polling interrupt routine that services interrupts and arranges to transfer the next byte of data to the device.

The subsections to follow describe these routines in more detail.

6.1. General Declarations in Driver

In addition to including a bunch of system header files, there are some data structures that the driver must define.

```
#include "../h/param.h"
#include "../h/buf.h"
#include "../h/file.h"
#include "../h/dir.h"
#include "../h/user.h"
#include "../h/uio.h"
#include "../machine/psl.h"
#include "../sundev/mbvar.h"

#include "sk.h"     /* file generated by config;
                       contains the definition of NSK */

#include "skreg.h"  /* register definitions */

#define SKPRI (PZERO-1) /* software sleep priority for sk */

#define SKUNIT(dev) (minor(dev))

struct buf skbufs[NSK];  /* static buffer headers for physio */

/* autoconfiguration-related declarations */
int skprobe(), skpoll();    /* kernel interface routines */
struct mb_device *skdinfo[NSK];
struct mb_driver skdriver = { skprobe, 0, 0, 0, 0, skpoll,
    sizeof(struct sk_reg), "sk", skdinfo, 0, 0, 0, 0,
    };

/* device state information -- global to driver */
struct sk_device {
    char soft_csr;      /* software copy of csr */
    struct buf *sk_bp;  /* current buf */
    int sk_count;       /* number of bytes to send */
    char *sk_cp;        /* next byte to send */
    char sk_busy;       /* true if device is busy */
} skdevice[NSK];
```

Here's a brief discussion on the declarations in the above example.

`sk.h`	file is automatically generated by `config`. It contains the definition of NSK, the number of sk devices configured into the system.
`SKPRI`	declaration declares the software priority level at which this device driver will sleep.
`SKUNIT`	macro is a common way of obtaining the minor device number in a driver. Study just about any device driver and you will find a declaration like this — it is a stylized way of referring to the minor device number. One reason for this is that sometimes a driver will encode the bits of the minor device number to mean things other than just the device number, so using the SKUNIT convention is an

easy way to make sure that if things change, the code will not be affected.

skbufs	array is necessary so that the driver will have its own `buf` headers to pass to the `physio()` routine. Character drivers should *never* use `buf` headers from the kernel's I/O queue. `physio()` will fill in certain fields (only a few, really) before calling *xx*`strategy()` with the `buf` structure as the argument.

There then follows a series of declarations, one for each of the autoconfiguration-related entry points into the device driver. In this driver, the only such entry points we use are `skprobe()` (which probes the Main Bus during system configuration) and `skpoll()` (the polling interrupt routine).

`skdinfo`	is an array of pointers to the `mb_device` structures that correspond to the driver's devices. The autoconfiguration process will initialize it during kernel boot time.
`skdriver`	is a definition of the `mb_driver` structure for this driver. An explanation of the fields in this structure and how they are initialized appears in the *Autoconfiguration-Related Declarations* section of this manual. This data structure is the major linkage to the kernel. It *must* be called *driver-name*`driver` where *driver-name* is the name of the device driver. `config` assumes that all device-driver structures have names in the form *driver-name*`driver`.
`sk_device`	is a definition of a structure, global to the driver, that holds driver-specific state information.

6.2. Autoconfiguration Procedures

Sun device drivers are tightly bound to the Sun autoconfiguration system. They assume, at compile time, that certain services have been provided for them by `config`, and they, in turn, provide boot-time hooks by which the kernel can determine if the actual system configuration matches that given in its `config` file.

There are, essentially, two autoconfiguration routines provided by the driver. The first is *xx*`probe()`, the second *xx*`attach()`. For more information, see the *Overall Kernel Context* section of this manual.

`probe()` Routine

There should be an *xx*`probe()` function in every driver. During the system boot each device entry in the config file generates a call to the *xx*`probe()` routine in the corresponding driver. *xx*`probe()` has three functions:

1. To determine if a device is present at the address indicated in the config file.
2 To determine if it's the expected type of device.
3. To notify the kernel of the system resources required for the device.

Under normal circumstances, addressing non-existent memory or I/O space on the Multibus or the VMEbus generates a bus error in the CPU. The kernel, however, supports checking for device existence with a set of functions designed to probe the address space, recover from possible bus errors, and return an indication as to whether the probe generated a bus error.

These functions are `peek()`, `peekc()`, `peekl()`, `poke()`, `pokec()`, and `pokel()`. They provide for accessing possibly non-existent addresses on the bus without generating the bus errors that would otherwise terminate the process trying to access such addresses. `peek()` and `poke()` read and write, respectively, 16-bit words ("shorts" on Sun2s and Sun3s, "half-words" on Sun-4s). `peekc()` and `pokec()` read and write 8-bit characters. In general, you will use the character routines for probing single-byte I/O registers. See the *Kernel Support Routines* appendix for details on these routines.

Having determined whether the device exists in the system, the *xx*`probe()` function returns either:

- the size (in bytes) of the device structure if it does exist. The kernel uses the value returned from `probe()` to reserve memory resources for that device. For both I/O-mapped and memory-mapped devices, *xx*`probe()` returns the total amount of space consumed by the device registers and memory.
- a value of 0 (zero) if the device does not exist.

Now we can write `skprobe()`:

```
/*ARGSUSED*/
skprobe(reg, unit)
    caddr_t reg;
    int unit;
{
    register struct sk_reg *sk_reg;
    register int c;

    sk_reg = (struct sk_reg *)reg;

    /* contact the device */
    c = peekc((char *)&sk_reg->sk_csr);
    if (c == -1 || (c != SK_ISTHERE))
        return (0);

    /* contact the device */
    if (pokec((char *)&sk_reg->sk_csr, SK_RESET))
        return (0);

    return (sizeof (struct sk_reg));
}
```

The *reg* argument is the purported address of the device, as given in the `config` file. The *unit* argument is only needed for controller drivers that must distinguish among multiple slave devices.

The *xx*`probe()` routine determines that the device actually exists, resets it to make sure that it's ready to go, and then returns the amount of bus space that it uses to the kernel autoconfiguration process. If *xx*`probe()` finds the device, the `md_alive` field in the device structure is set to 1, otherwise it's set to 0. `md_alive` is subsequently used by other driver (and kernel) functions to check that the device was probed successfully at startup time. (These routines can also check the device's position in the driver's *xx*`dinfo()` array (if it has one) to see if it's been initialized).

`attach()` Routine

The second autoconfiguration routine is *xx*`attach()`. The purpose of *xx*`attach()` is to do device-specific initialization. Such initialization may include the issuing of commands to the actual device hardware, for example, the disabling of its interrupts, or it may be entirely confined to the initialization of local device-specific structures. It's up to the driver what kind of initialization is done in *xx*`attach()`.

The Skeleton device is artificially simple, and it requires no initialization besides the assignment of `SK_RESET` into its control/status register. This assignment, as you will note, has already been done in `skprobe()`, where it serves as a doublecheck on the correct installation of the device. Since no further initialization is necessary, the Skeleton driver needs no `attach()` routine.

6.3. `open()` and `close()` Routines

During the processing of an `open()` call for a special file, the system always calls the device's *xx*`open()` routine to allow for any special processing required (rewinding a tape, turning on the data-terminal-ready lead of a modem, and so on). However, the *xx*`close()` routine is called only when the last process closes a file, that is, when the i-node table entry for that file is being deallocated. Thus it is not feasible for a device driver to maintain, or depend on, a count of its users, although it is quite simple to implement an exclusive-use device that can't be reopened until it has been closed.

`skopen()` is quite straightforward. It's called with two arguments, namely, the device to be opened, and a flag indicating whether the device should be opened for reading, writing, or both. The first task is to check whether the device number to be opened actually exists — `skopen()` returns an error indication if not. The second check is whether the open is for writing only. Since the Skeleton device is write only, it's an error to open it for reading. If all the checks succeed, `skopen()` enables interrupts from the device, and then returns zero as an indication of success. Here's the code for `skopen()`:

```
skopen(dev, flags)
    dev_t dev;
    int flags;
{
    register int unit = SKUNIT(dev);
    register struct mb_device *md;
    register struct sk_reg *sk_reg;

    md = skdinfo[unit];

    if (unit >= NSK || md->md_alive == 0)
        return (ENXIO);
    if (flags & FREAD)
        return (ENODEV);

    sk_reg = (struct sk_reg *)md->md_addr;

    /* enable interrupts */
    skdevice[unit].soft_csr = SK_ENABLE;

    /* contact the device */
    sk_reg->sk_csr = skdevice[unit].soft_csr;

    return (0);
}
```

The first `if` statement checks if the device actually exists. The first clause

```
(unit >= NSK)
```

is necessary because, as root, someone could make a special file that has a minor device number greater than NSK then try to open it. This actually isn't unusual, many /dev directories have entries for devices that are not really installed. The second clause tests to see if the *probe* routine found the device. Note the use of the SKUNIT macro to obtain the minor device number — we discussed this earlier on. Also note that we're maintaining a copy

```
(skdevice[unit].soft_csr)
```

of the control/status register in memory. Each time we write the register we will do so first in memory and then in the actual hardware register. We will do this doggedly, to make the point that we must protect ourselves from the potential side effects of inadvertent calculations within registers. For example

```
csr &= ~SK_ENABLE
```

has the side effect of reading the csr register — and patterns read from this register are *not* always identical to those written into it. (For more information, see the *Hardware Peculiarities to Watch Out For* section of the *Hardware Context* chapter).

`skclose()` is quite straightforward, since all it does is disable interrupts:

```
/*ARGSUSED*/
skclose(dev, flags)
    dev_t dev;
    int flags;
{
    register int unit = SKUNIT(dev);
    register struct mb_device *md;
    register struct sk_reg *sk_reg;

    md = skdinfo[unit];

    /* disable interrupts */
    sk_reg = (struct sk_reg *)md->md_addr;
    skdevice[unit].soft_csr &= ~SK_ENABLE;

    /* contact the device */
    sk_reg->sk_csr = skdevice[unit].soft_csr;
}
```

`skclose()` could in fact be more complicated than this. It could, for example:

- deallocate resources that were allocated for the device being closed, or
- shut down the device itself, for example by signaling a port to hang up.

6.4. `read()` and `write()` Routines

The Skeleton device is write-only, but this discussion would apply equally to reading in such a non-tty oriented character device.

When a *read* or *write* takes place, the user's arguments — as well as some system-maintained information about the file to which the I/O operation is to be performed — are used to initialize two structures — `uio` and `iovec` — that are used for character I/O. The fields of greatest interest within these structures are `iovec.iov_base`, `iovec.iov_len`, and `uio.uio_offset` which respectively contain the (user) address of the I/O target area, the byte-count for the transfer, and the current location in the file. If the file referred to is a character-type special file, the appropriate *xx*`read()` or *xx*`write()` routine is called — this routine is responsible for transferring data and updating the count and current location appropriately as discussed below.

For most non-tty devices, *xx*`read()` and *xx*`write()` call *xx*`strategy()` through the system `physio()` routine. `physio()` ensures that the user's memory space is locked into core (not paged out) for the duration of the data transfer. It also provides an automated way of breaking a large transfer into a series of smaller, more manageable ones. Note that character drivers that use `physio()` must declare an array of `buf` structures, one for each of their devices (here the array is named `skbufs`). By doing so they avoid any need to use the kernel's buffer cache, which is provided for the use of system block-structured devices.

xx`write()` differs from *xx*`read()` only in the value of the flag it passes to `physio()`. `skwrite()` looks like this:

```
skwrite(dev, uio)
    dev_t dev;
    struct uio *uio; See note on the uio structure below
{
    int unit = SKUNIT(dev);

    if (unit >= NSK)
        return (ENXIO);
    return (physio(skstrategy, &skbufs[unit], dev,
        B_WRITE, skminphys, uio));
}
```

The `skminphys()` routine is called by `physio` to determine the largest reasonable block size to transfer at once. If the user requests a larger transfer, `physio()` will call `skstrategy()` repeatedly, requesting no more than this block size each time. This is important when DVMA transfers are done. (DVMA is covered in more detail below). The reasoning is that only a finite amount of address space is available for DVMA transfers and it is not reasonable for any device to tie up too much of it. A disk or a tape might reasonably ask for as much as 63 Kilobytes; slow devices like printers should only ask for one to four Kilobytes since they will tie up the resource for a relatively long time. Here's the `skminphys()` routine.

```
skminphys(bp)
    struct buf *bp;
{

    if (bp->b_bcount > MAX_SK_BSIZE)
        bp->b_count = MAX_SK_BSIZE;
}
```

Note that if you don't supply your own `minphys()` routine, you place the name of the system supplied `minphys()` routine, whose name is `minphys()`, as the argument to `physio()` in its place, and the system supplied `minphys()` routine gets used instead. This is not always a good thing, however, for the system routine divides an I/O operation into 63K chunks, and this can be too large for optimum system performance when the device in question is slow (like a printer).

Some Notes About the UIO Structure

When the system is reading and writing data from or to a device, the `uio` structure is used extensively (see `/usr/include/sys/uio.h` for more information). The `uio` structure is generalized to support what is called *gather-write* and *scatter-read*. That is, when writing to a device, the blocks of data to be written don't have to be contiguous in the user's memory but can be in physically discontiguous areas. Similarly, when reading from a device into memory, the data comes off the device in a continuous stream but can go into physically discontiguous areas of the user's memory. Each discontiguous area of memory is described by a structure called an `iovec` (I/O vector). Each `iovec` contains a pointer to the data area to be transferred, and a count of the number of bytes in that area. The `uio` structure describes the complete data transfer. `uio` contains a pointer to an array of these `iovec` structures. Thus when you want to write a number of physically discontiguous blocks of memory to a device, you can set up an array of `iovec` structures, and place a pointer to the start of the array in the `uio` structure. In the simplest case, there's just one block of data to be transferred, and the `uio` structure is quite simple. Note that `physio()` will call the *strategy* routine at least once for each `iovec` contained by the `uio` structure.

6.5. Skeleton `strategy()` Routine

xx`strategy()` is called by `physio()` after it has locked the user's buffer into memory. The name *strategy* originated in the world of disk drivers, and implied that the routine could be clever about queuing I/O requests (for example, by disk address) so as to minimize time wasted by the disk. The `skstrategy()` routine has no such problems, since it doesn't queue I/O requests for a random-access device. Still, a number of tasks remain — `skstrategy()` must check that the device is ready, initiate the data transfer, and wait for its completion to be signaled by the interrupt routine. Note that `skstrategy()` can safely assume that `physio()` has properly initialized a number of variables — here we will assume that the `b_dev` field in the `buf` has been set to contain the device number.

```
skstrategy(bp)
    register struct buf *bp;
{
    register struct mb_device *md;
    register struct sk_device *sk;
    int s;

    md = skdinfo[SKUNIT(bp->b_dev)];
    sk = &skdevice[SKUNIT(bp->b_dev)];

    s = splx(pritospl(md->md_intpri)); /* begin critical section */
    while (sk->sk_busy)
        sleep((caddr_t) sk, SKPRI);

    /* set up for first I/O operation */
    sk->sk_busy = 1;
    sk->sk_bp = bp;
    sk->sk_cp = bp->b_un.b_addr;
    sk->sk_count = bp->b_bcount;
    skstart(sk, (struct sk_reg *)md->md_addr);

    (void) splx(s); /* end critical section */
}
```

xx`strategy()` doesn't actually do any I/O. It just insures that the device is not busy, (by sleeping on the address of a data structure that is global to the driver) sets up for the first I/O operation and then calls *xx*`skstart()` to get things rolling. The critical section is necessary because *xx*`strategy()` is trying to acquire the device on behalf of one, and only one, user process.

6.6. Skeleton `start()` Routine

xx`start()` is actually responsible for getting the data to or from the device. `skstart()` is called once directly from `skstrategy()` to get the very first byte out to the device. After that, it is assumed that the device will interrupt every time it is ready for a new data byte, and so `skstart()` is thereafter called from `skintr()`. Here is one possible `skstart()` routine:

```
skstart(sk, sk_reg)
    struct sk_device *sk;
    struct sk_reg *sk_reg;
{

    sk_reg->sk_data = *sk->sk_cp++;

    if (--sk->sk_count > 0) {
        sk->soft_csr = SK_ENABLE;

        /* contact the device */
        sk_reg->sk_csr = sk->soft_csr;
    }
}
```

This routine will work, but not very efficiently. There's a lot of overhead in taking a device interrupt on every character. Since we know that the device can accept characters very quickly, it would be much more efficient to give the characters quickly, and thus avoid generating unnecessary interrupts. *xx*`start()` should take advantage of device-specific characteristics to win efficiency enhancements of this type. It can wait for characters, check for ready, etc — here, we will just check after each character and give another one if the device is ready for it. Here's the new, more efficient `skstart()` routine.

```
skstart(sk, sk_reg)
    struct sk_device *sk;
    struct sk_reg *sk_reg;
    {

    while(sk->sk_count > 0) {   /* still more characters */
        sk_reg->sk_data = *sk->sk_cp++;
        sk->sk_count--;

        /* stop giving characters if device not ready */
        /* Note: the softcopy isn't needed for reads */
        /* contact the device */

        /* DELAY(10) might go here */

        if (!(sk_reg->sk_csr & SK_DEVREADY))
            break;
    }

    /* error-retry logic would go here */

    /* still more characters */
    if (sk->sk_count > 0) {
        sk->soft_csr = SK_ENABLE;

        /* contact the device */
        sk_reg->sk_csr = sk->soft_csr;
    } else {
        /* special case: finished command without taking any interrupts! */

        /* disable interrupts */
        sk->soft_csr = 0;

        /* contact the device */
        sk_reg->sk_csr = sk->soft_csr;
        sk->sk_busy = 0;

        /* free device to sleeping strategy routine */
        wakeup((caddr_t) sk);

        /* free buffer to waiting physio */
        iodone(sk->sk_bp);
    }
}
```

We give characters to the device as long as there are more characters and the device is ready to receive them. If we run out of characters, we disable interrupts to keep the device from bothering us and call `iodone()` to mark the buffer as done.

It may be that the device is not quite quick enough to take a character and raise the `SK_DEVREADY` bit in the time we can decrement the counter. If so, it would be very worthwhile to busy wait for a short time. The reasoning is that while busy waiting is a waste, servicing an interrupt costs lots more CPU time, and if busy waiting works fairly often it is a big win. There is a macro `DELAY()` that takes an integer argument which is approximately the number of microseconds to delay, so we could add

```
DELAY(10);
```

at the top of the `while` loop. Clearly this is an area where experimentation with the real device is called for.

6.7. `intr()` and `poll()` Routines

Each device should have appropriate interrupt-time routines. When an interrupt occurs, it is transformed into a C-compatible call on the device's interrupt routine. After the interrupt has been processed, a return from the interrupt handler returns from the interrupt itself.

The address of the polling interrupt routine for a particular device driver is contained in the per-driver (that is, `mb_driver`) data structure for that device driver. It is installed there during the kernel configuration process based upon information in the config file.

Since (on Multibus machines) devices typically need to share interrupt levels, it's the specific driver's responsibility to determine if the interrupt is intended for it or not. The driver does so by providing a polling interrupt routine that queries the interrupt state of each of its devices in turn — if a driver doesn't provide such a routine, it won't work correctly on a Multibus machine. Polling interrupt routines that determine that an interrupt belongs to one of their devices must notify the kernel to that effect (after servicing the interrupt) by returning a non-zero value. If a polling interrupt routine determines that an interrupt is *not* from one of its devices, it must return a zero value.

It's expected that the device actually indicates when it's interrupting. If there are any more bytes to transfer, the interrupt routine calls *xx*`start()` to transfer the next byte. If there are no more bytes to transfer, the interrupt routine disables the interrupt (so that the device won't keep interrupting when there's nothing to do), and finishes up by calling `iodone()`. (`iodone()`, incidentally, is another of the mechanisms provided primarily for block drivers). Here are the interrupt routines for the Skeleton driver:

```
skpoll()
{
    register struct sk_reg *sk_reg;
    int serviced, i;

    serviced = 0;
    for (i = 0; i < NSK; i++) { /* try each one */
        sk_reg = (struct sk_reg *)skdinfo[i]->md_addr;

        /* contact the device */
        if (sk_reg->sk_csr & SK_INTR) {
            serviced = 1;
            skintr(i);
        }
    }
    return (serviced);
}
```

```
skintr(i)
    int i;
{
    register struct sk_reg *sk_reg;
    register struct sk_device *sk;

    sk_reg = (struct sk_reg *)skdinfo[i]->md_addr;
    sk = &skdevice[i];

    /* check for an I/O error */

    /* contact the device */
    if (sk_reg->sk_csr & SK_ERROR) {

        /* error-retry logic would go here */

        printf("skintr: I/O error0);
        sk->sk_bp->b_flags |= B_ERROR;
    }

    /* I/O transfer completed */
    if ((sk->sk_bp->b_flag & B_ERROR) != 0 ||
        sk->sk_count == 0) {

        /* clear interrupt */
        sk->soft_csr = 0;

        /* contact the device */
        sk_reg->sk_csr = sk->soft_csr;
        sk->sk_busy = 0;

        /* free device to sleeping strategy routine */
        wakeup((caddr_t) sk);
        /* free buffer to waiting physio */
        iodone(sk->sk_bp);
    } else
        skstart(sk, sk_reg);
}
```

`skintr()` checks the hardware for an error every time it's called, and upon finding an error, calls `printf()`, flags the error in the I/O buffer and then returns. Note that:

- `skintr()` needs the buffer header associated with the failed transfer so that it can indicate the error in its `b_flags` field.
- A retry attempt could be made before giving up and taking the error return. Whether or not this is advisable is entirely dependent on the specific device and error characteristics.
- The error return aborts the I/O request that produced the error and then places both the device and the driver in their normal idle states.

6.8. `ioctl()` Routine

xx`ioctl()` is used to perform any tasks that can't be done by *xx*`open()`, *xx*`close()`, *xx*`read()`, or *xx*`write()`. Typical applications are: "what is the status of this device", or "go into mode X". The Skeleton device, as we've defined it here, is modeless and has no such special functions so we don't have an *xx*`ioctl()` routine. (Though we will add one below in a variation of the Skeleton driver that supports a form of asynchronous I/O). For details about driver *xx*`ioctl()` routines, and the other driver routines, see the *Summary of Device Driver Routines* appendix.

6.9. Skeleton Driver Variations

The Skeleton I/O board isn't particularly realistic, but is does serve to illustrate the construction of a basic character driver. In this section, we will propose some variations on the basic device, each designed to illustrate a useful technique.

DMA Variations

Devices that are capable of doing DMA are treated differently than the Skeleton device we've been working with so far. Let's assume that we have a new version of the Skeleton board; call it the Skeleton II. It can do DMA transfers and we want to use this feature since it is much more efficient.

NOTE *DMA is different on the Sun386i system. For information about it, see the* `dma_setup()` *and* `dma_done()` *routines in the Kernel Support Routines appendix.*

Multibus or VMEbus DVMA

The Sun processor board is always listening to the Multibus or VMEbus for memory references. When there is a request to read or write any address in the DVMA space (see the *Sun Main-Bus DVMA* section of the *Hardware Context* chapter for more information) the DVMA hardware adds a machine-specific offset to the address to find the location in kernel virtual memory that contains the device RAM being used in the transfer.

On Sun-2 Multibus machines, DVMA space consists of all addresses between 0 and `0x3FFFF`. On Sun-2 VMEbus machines, it consists of all addresses between `0x0` and `0xFFFFF`. Upon encountering one of these addresses, the DVMA hardware adds `0xF00000` to get the system virtual address of the device RAM.

On the Sun-3, the DVMA space is defined by the address range `0x0` to `0xFFFFF` for 24-bit or 32-bit addressing; its system virtual address is `0xFF00000`.

On the Sun-4, the DVMA space is defined by the same address range used on the Sun-3, `0x0` to `0xFFFFF` for 24-bit or 32-bit addressing. Its system virtual address, however, is `0xFFF00000`.

If you wish to do DMA over the Main Bus, you must make entries in the kernel memory map to map your device's RAM into the appropriate DVMA space. As you might expect, there are subroutines to help with this chore. `mbsetup()` sets up the kernel memory map and `mbrelse()` clears entries in it to release DVMA space. Note that all Sun DMA occurs between the bus and kernel virtual address space — if you wish to do DMA directly into a user buffer, you will have to first map that buffer into kernel space, then pass it to `mbsetup()` to map it into DVMA space.

A DMA Skeleton Driver

The addition of DMA to the capabilities of the device opens up several new options. For the moment, consider only the changes necessary to switch the driver over to DMA-style I/O. These changes turn out to be surprisingly straightforward. First we will extend the `sk_reg` structure which defines the device registers. We will assume that the Skeleton II board is a bus-master which supports 20-bit transfers, and that the following structure overlays its registers.

```
struct sk_reg {
    char sk_data;           /* 01: Data Register */
    char sk_csr;            /* 00: command(w) and status(r) */
    short sk_count;         /* bytes to be transferred */
    caddr_t sk_addr;        /* 20-bit DMA address */
};
```

Next we assume that bit 5 in the csr is set to initiate a DMA transfer.

```
#define SK_DMA  0x10      /* Do DMA transfer */
```

and a definition of the maximum DMA transfer for `skminphys()`.

```
#define MAX_SK_BSIZE 4096     /* DMA transfer block */
```

And we must add another element to the `sk_device` structure for use by `mbsetup()` and `mbrelse()`. (The alternative would be to use the `mc_mbinfo` structure in the `mb_ctlr` structure, but since we don't use that structure for anything else, this seems more reasonable):

```
int sk_mbinfo;
```

Now we change `skstrategy()` to use the DMA feature.

```
skstrategy(bp)
    register struct buf *bp;
{
    register struct mb_device *md;
    register struct sk_reg *sk_reg;
    register struct sk_device *sk;
    int s;

    md = skdinfo[SKUNIT(bp->b_dev)];
    sk_reg = (struct sk_reg *)md->md_addr;
    sk = &skdevice[SKUNIT(bp->b_dev)];

    s = splx(pritospl(md->md_intpri));   /* begin critical section */
    while (sk->sk_busy)
        sleep((caddr_t) sk, SKPRI);
    sk->sk_busy = 1;
    sk->sk_bp = bp;

    /* this is the part that is changed */

    /* grab bus resources */
    sk->sk_mbinfo = mbsetup(md->md_hd, bp, 0);

    /* plug the remainder */
    sk_reg->sk_count = bp->b_bcount;

    /* plug bus transfer address */
    sk_reg->sk_addr = (caddr_t)MBI_ADDR(sk->sk_mbinfo);

    /* make sure we didn't overrun the address space limit */
    if (sk_reg->sk_addr > (caddr_t) 0x000FFFFF) {
        printf("sk%d: ", sk_reg->sk_addr);
        panic("exceeded 20 bit address");
    }

    sk->soft_csr = SK_ENABLE | SK_DMA;
    sk_reg->sk_csr = sk->soft_csr; /* contact the device */

    /* end of DMA-related changes */

    (void) splx(s);      /* end critical section */
}
```

There are a number of details here that are worth noting:

- `skstart()` is no longer needed and may be completely eliminated.
- The return value from `mbsetup()` is being saved for use in calls to `MBI_ADDR()` and `mbrelse()`.
- The 32-bit address returned by `MBI_ADDR()` is being tested to ensure that it doesn't exceed the 20-bits limits of the device. (This wouldn't be necessary if the address was sure to be in the DVMA transfer area, which always

ends at `0xFFFFF` or below. However, the transfer address can also be elsewhere in the VMEbus address space).

- All the I/O now is started by `skstrategy()` and continues until `skpoll()` is called — thus we can delete the `sk_cp` and `sc_count` fields from the `sk_device` structure.
- There's no longer any need to check the count and sometimes call `skstart()`. Instead, `iodone()` is always called and `physio()` is relied upon to proceed with the transfer. Note that, with `skstart()` eliminated, the call to `wakeup()`, as well as the clearing of `sk_busy`, have been moved to `skintr()`.
- Finally, `skintr()` needs to free up the Main Bus resources, so it will call `mbrelse()`.

Here are the new `skintr()` and `skpoll()` routines:

```
skintr(i)
    int i;
{
    register struct mb_device *md;
    register struct sk_reg2 *sk_reg;
    register struct sk_device *sk;

    md = (struct mb_device *)skdinfo[i];
    sk_reg = (struct sk_reg2 *)md->md_addr;
    sk = &skdevice2[i];

    /* check for an I/O error */
    if (sk_reg->sk_csr & SK_ERROR) { /* contact the device */

        /* error-retry logic would go here */

        printf("skintr: I/O error\n");
        sk->sk_bp->b_flags |= B_ERROR;
    }

    /* this is the part that changed */
    sk->soft_csr = 0;     /* clear interrupt */
    sk_reg->sk_csr = sk->soft_csr;
    mbrelse(md->md_hd, &sk->sk_mbinfo);

    sk->sk_busy = 0;
    wakeup((caddr_t)sk); /* free device to sleeping strategy routine */
    iodone(sk->sk_bp);    /* free buffer to waiting physio */
}
```

```
skpoll()
{
    register struct mb_device *md;
    register struct sk_reg *sk_reg;
    int serviced, i;

    serviced = 0;
    for (i = 0; i < NSK; i++) {
        md = (struct mb_device *)skdinfo[i];
        sk_reg = (struct sk_reg *)md->md_addr;
        if (sk_reg->sk_csr & SK_INTR) {
            serviced = 1;
            skintr(i);
        }
    }
    return (serviced);
}
```

Variation with ``Asynchronous I/O'' Support

In this next section, we will assume that we want to further modify the Skeleton driver to support ``asynchronous I/O''. This may, at first sight, seem an odd thing to do, for asynchronous I/O is most commonly used for network and serial-line devices that have little in common with the Skeleton device. In actual fact, however, asynchronous I/O is *not* limited in application to such devices — its purpose is to support user processes which need to avoid blocking during I/O operations, and such functionality is of interest for serial lines, sockets, STREAMS and various character devices.

First, note that the term ``asynchronous I/O'' is used, in the UNIX world, to indicate two separate mechanisms. In practice, these mechanisms are closely related, and both of them will be be covered in this section:

- The first is ``non-blocking I/O''. This is a type of I/O which, when incapable of immediately proceeding to completion, notifies its user process of this fact rather than simply going to `sleep()`. It thus gives the user process a choice of responses.

  In the UNIX system, non-blocking I/O is traditionally provided by the `select()` system call, which allows a user process to query a device to see if it's ready before making a `read()` or `write()` request to it, and thus to avoid being blocked. (It should be noted that `select()` isn't really non-blocking I/O proper. It's better thought of as an alternative to device polling, which can waste considerable CPU time).

- The second UNIX asynchronous I/O mechanism is best called ``asynchronous notification''. With this mechanism available, the user process no longer needs to keep trying an I/O operation until it succeeds, because the driver will `signal()` it (with a `SIGIO`) when one of its I/O channels clears. The code necessary to support such asynchronous notification is closely related to that necessary to support `select()`, and it should routinely be provided at the same time as `select()` support.

Select Routines

The Skeleton driver hasn't really been defined as a device that we would expect to have a `select()` routine. Such routines are most useful for devices which aren't always ready, and since we've defined the Skeleton device as being write only and arbitrarily fast, we wouldn't expect it to clog. Still, for the purposes of this example, we will assume that the Skeleton board is sufficiently slow that it's reasonable to have its driver support `select()`.

`select()` is more typically used in serial-line drivers which are multiplexed between multiple lines. Before reading, for example, a terminal's keyboard, such drivers need to ensure that there are characters waiting. If they didn't, they would block so often that their overall performance would be unacceptable.

`select()` works by providing user processes with a means of determining if I/O is possible on a given file descriptor. Alternatively, it has a multiplexing feature that makes it possible to determine which of a set of specified descriptors is ready to go. It can be told to return immediately, or to block the calling process until at least one descriptor is ready. A timeout argument can be specified to keep the process from blocking forever, or to allow the process to periodically do something else. See `select(2)` for details.

The driver's `select()` routine may or may not support the full functionality of the `select()` system call. The minimum that it can reasonably do is allow the user program to poll the specified device to determine if it's ready:

```
skselect(dev, rw)
    dev_t dev;
    int rw;
{
    register struct mb_device *md;
    register struct sk_reg *sk_reg;
    int s = spl5();

    md = skdinfo[SKUNIT(bp->b_dev)];
    sk_reg = md->md_addr;

    /* Check if the device is ready */
    if (sk_reg->sk_csr & SK_DEVREADY) {
        (void) splx(s);
        return (1);
    }
    (void) splx(s);
    return (0);
}
```

Note that, in this example, the `rw` flag has been ignored because the Skeleton device is write only. If, however, it were a read/write device, `skselect()` would switch on `rw`, and do a separate readiness test for each of the `READ` and `WRITE` cases. Throughout this example we will show only write cases: read cases would be handled identically.

To extend `skselect()` to allow user processes to block for specified periods of time (or, for that matter, indefinitely) while waiting for an OK to proceed with an I/O operation, more must be done. To begin with, we must add two fields to the `sk_device()` structure. Both of them must be initialized to 0.

```
struct sk_device {
    ...
    struct proc *sk_wsel;     /* user proc structure */
    int sk_state;             /* select state flag */
    };
```

We also need the flag

```
#define SK_WCOLL    0x01
```

which will be used to indicate that a write-select collision has occurred, that is to say, that more than one process has attempted to select the device.

Then, `skselect()` must be changed, as follows:

```
skselect(dev, rw)
    dev_t dev;
    int rw;
{
    register struct mb_device *md;
    register struct sk_reg *sk_reg;
    register struct sk_device *sk;

    int s = spl5();

    md = skdinfo[SKUNIT(dev)];
    sk_reg = md->md_addr;
    sk = &skdevice[SKUNIT(dev)];

    /* Check if the device is ready */
    if (sk_reg->sk_csr & SK_DEVREADY) {
        (void) splx(s);
        return (1);
    }

    /* Here's the new code */
    if (sk->sk_wsel &&
       (sk->sk_wsel->p_wchan == (caddr_t) &selwait))
        sk->sk_state |= SK_WCOLL;
    else
        sk->sk_wsel = u.u_procp;

    (void) splx(s);
    return (0);
}
```

`selwait()`, an external integer imported via `<sys/systm.h>`, is the "channel" which the `select()` system call, and only the `select()` system call, uses when it calls `sleep()`.

If the device is ready to go, `skselect()` behaves just as it did above: it returns immediately with a 1. If, however, the device isn't ready, a check is made to see if it has already been selected. If it hasn't been, the field `sk_wsel` is set to point to the `proc` structure of the process doing the select. In effect, we're remembering the first process to select the device. If no other processes select the same device, this structure will later be used as a "fast path" to the selecting process.

If, however, `skselect()` finds that `sk_wsel` has already been set, the test:

```
(sk->sk_wsel->p_wchan == (caddr_t) &selwait)
```

is made to see if the process indicated by `sk->sk_wsel` is sleeping as a result of a call to `select()`. If it is, the code

```
sk->sk_state |= SK_WCOLL;
```

is executed to indicate that a select "collision" has occurred, that is, that a second (or third, etc.) process attempted to select the device while the first process was still waiting for it to become available.

The rest of the select-related code is executed at interrupt time, so it goes into `skintr()`. One clean way of inserting it is to create a new routine, `skwakeup()`, and to call it from `skintr()` instead of calling `wakeup()`. (See the non-DMA version of `skintr()`, above):

```
skwakeup(sk)
    register struct sk_device *sk;
{
    if (sk->sk_wsel) {  /* select is pending */

        /* wake up the process */
        selwakeup(sk->sk_wsel, sk->sk_state & SK_WCOLL);

        /* reset the select flags */
        sk->sk_state &= ~SK_WCOLL;
        sk->sk_wsel = 0;
    }
    wakeup((caddr_t) sk);
}
```

`selwakeup()` thus receives a `NULL` second parameter unless a select collision occurred. If such a collision did occur, *all* processes which are sleeping as a result of a `select()` (any select) are awakened by a call to `wakeup()` on the `selwait()` channel. Most of them will just go back to sleep, and the ones that don't will race for the device. This isn't very efficient, but it doesn't happen very often. Usually, the device will be selected by a single process, and the `proc` structure will be used to wake only that process.

Note that `selwakeup()` does nothing if `sk->sk_wsel` is 0, or if there are no processes sleeping on `selwait()`. Thus, if a process has called `select()`, but not gone to sleep (because the device was immediately ready) the subsequent interrupt will simply reset the flags.

Adding Asynchronous Notification

If the driver is to support asynchronous notification as well as `select()`, a bit more is necessary. First, a new flag is necessary to indicate that the user has requested asynchronous notification:

```
#define SK_ASYNC    0x02
```

And a new field is necessary in the `sk_device` structure, which now becomes:

```
struct sk_device {
    . . .
    struct proc *sk_wsel;      /* user proc structure */
    int sk_state;              /* select state flag */
    short p_pgrp;              /* user process group leader */
    };
```

The new field, `p_pgrp` must, like the others, be initialized to 0. And `p_pgrp` must be initialized in `skopen()` to indicate the process group leader of the user process opening the device:

```
    . . .
    if (sk->p_pgrp == 0)
        sk->p_pgrp = (u.u_procp)->p_pid;
```

Next, we must provide a way for the user process to request that the driver enable asynchronous notification. Of course it would be possible for it to always operate in asynchronous mode, but then user processes would constantly get sent `SIGIO` signals by the driver, whether they expected them or not. Besides, if the Skeleton driver has multiple *modes*, we must introduce an `skioctl()` routine to toggle them, and that gives us an opportunity to discuss *ioctl* routines. Actually, there are potentially three system calls that can be used to put a driver into asynchronous mode, or, for that matter, into any mode. The most common of these is `ioctl(2)`, and it is it that we will show here. Note, though, that the other two possibilities are `fcntl(2)` and `open(2)`.

Adding an **ioctl()** routine

The first step in introducing an `ioctl()` routine is to define the macros which user processes will use to issue commands to the device and its driver. (For details, see the discussion of `ioctl()` routines in the *Summary of Device Driver Routines* appendix to this manual).

In the case of `skioctl()`, these macros are few and simple, for `skioctl()` will only toggle the driver mode between synchronous and asynchronous. There's no need for the `ioctl()` macros to either ship data from, or return it to, the user program.

`ioctl`– related command codes are exported to user processes by means of macros kept, by convention, in `/usr/include/sys`. In the case of the

Skeleton driver, only two macros are necessary, and we will put them into `/usr/include/sys/skcmds.h`:

```
#define SKSETSYNC    _IO(k,0)
#define SKSETASYNC   _IO(k,1)
```

The `_IO` macro is the simplest of the `ioctl()` macros, being intended for purposes like this, where no argument data need be transferred. Here, all that's necessary is to define a convention by which 0 indicates synchronous mode (the default) and 1 indicates asynchronous mode. Note the first parameter, 'k'. It's used, quite arbitrarily, to identify the `ioctl()` to be vectored to the Skeleton driver. It's only necessary to choose a letter that is not already in use by another driver.

The additions to the driver are very simple. First, it must include the file containing its control macros:

```
#include <sys/skcmds.h>
```

Then, in `skioctl()` it simply takes the information encoded by the `_IO` macro to toggle the driver's state:

```
skioctl(dev, cmd, data, flag)
    dev_t dev;
    int cmd;
    caddr_t data;
    int flag;
{
    register struct sk_device *sk;
    sk = &skdevice[SKUNIT(dev)];

    switch (cmd) {

    case SKSETSYNC:
        sk->sk_state &= ~SK_ASYNC;
        break;
    case SKSETASYNC:
        sk->sk_state |= SK_ASYNC;
        break;
    }
}
```

That's it. And now that `skioctl()` can set the `SK_ASYNC` flag, `skwakeup()` can reasonably test for it and, if it's set, call `gsignal()` to send the `SIGIO` signal to the user process group. Note that the `SK_ASYNC` signal must be cleared after calling `gsignal()`.

```
skwakeup(sk)
    register struct sk_device *sk;

    if (sk->sk_wsel) {   /* select is pending */

        /* wake up the process */
        selwakeup(sk->sk_wsel, sk->sk_state & SK_WCOLL);

        /* reset the select flags */
        sk->sk_state &= ~SK_WCOLL;
        sk->sk_wsel = 0;
    }
    if (sk->sk_state & SK_ASYNC) {
        gsignal(sk->p_pgrp, SIGIO);
        sk->sk_state &= ~SK_ASYNC;
        }
    wakeup((caddr_t) sk);
}
```

The final step in adding a `select()` routine to a driver is to edit the kernel `conf.c` file, and to plug the name of the new `select()` routine into the `cdevsw` structure in the place of the “nodev” or “seltrue” that is already there.

7

Configuring the Kernel

Configuring the Kernel **139**

7.1. Background Information 139
7.2. An Example 141
7.3. Devices that use Two Address Spaces 145
7.4. Adding and Removing Loadable Drivers 146

7

Configuring the Kernel

7.1. Background Information

In this chapter, we will assume that you've written your driver. The next step, obviously, is to build a kernel that includes your new driver. This process isn't difficult; Sun systems support easy kernel configuration, even without access to system source code. If the driver is a loadable driver then the kernel is not rebuilt and therefore the discussion of rebuilding the kernel does not apply. In this case, see the *Loadable Drivers* section of the *Driver Development Topics* chapter.

In heterogeneous server/client environments, kernels must be configured in fairly general ways. For one thing, they must work on both Multibus and VMEbus machines, for another, they have to tolerate normal variations among system devices (e.g. client Ethernet boards may be made by either 3COM or Sun). The `GENERIC` *config file thus contains configuration lines for all common devices for both bus types. However, if you're configuring a kernel for a known system, you need not carry around extraneous options — you can tailor your configuration file as appropriate and thus get a smaller (by 100 kilobytes or more!) and more efficient kernel.*

For additional information on kernel configuration, see the *Adding Hardware to Your System* section of *Network Programming* and the `config(8)` man page. (Incidently, `config` is found in the `/usr/etc/` directory — so make sure that your path includes `/usr/etc` before proceeding).

First, a simple distinction. If your kernel already contains a certain driver, and you're simply installing a corresponding device, you will only need to edit the kernel config file — all of the installation specific information about devices themselves is contained in this file. If, however, you will be adding a new driver to the kernel, you will need to edit some additional files:

- The first of these is `/usr/sys/sun/conf.c`, a C-language source-code file which contains the definitions of the switches `cdevsw` and `bdevsw`, as well as a bit of initialization infrastructure for the installed devices.
- The second is either `/usr/sys/sun2/conf/files`, `/usr/sys/sun3/conf/files`, `/usr/sys/sun4/conf/files`, or `/usr/sys/sun386/conf/files`, (depending upon the type of your machine). This file tells `config` where to find the source code for the kernel and its drivers.

The discussion in this chapter concerns `config`*, a utility program that is used in configuring kernels and initializing the kernel/driver interface structures.* `config` *is altogether different from the autoconfiguration process, sometimes called* `autoconfig`*, which is built into the initialization pass of the SunOS kernel, and thus run at system boot time. Autoconfiguration completes the run-time driver environment initialization that* `config` *begins, for example by checking that the devices indicated as present in the kernel config file are actually present in the running system. Autoconfiguration is discussed in much greater detail in the Overall Kernel Context chapter of this manual.*

`config`'s goal is to output a set of files that can be directly used to configure a new kernel. The purpose of the configuration may simply be to install a device (for which the kernel already contains a driver) or it may be to integrate a new device and its driver. The kernel configuration system learns of new devices by way of entries in the config file, whereas new drivers are indicated by editing one or all of the files `conf.c`, `/usr/conf.common/files_cmn` and `/usr/sys/sun[234]/conf/files` (or `/usr/sys/sun386/conf/files`). The files output by `config` are used in the construction of the new kernel, but so are others, notably `conf.c` itself.

- `ioconf.c` — the major input to the autoconfiguration process. It contains arrays of *mbvar* structures — `struct mb_ctlr mbcinit[]` and `struct mb_device mbdinit[]` — that have been initialized on the basis of the device and controller information in the config file. (Incidently, the order of the device declarations in the config file will determine the order of the structures in `ioconf.c`, and thus the order in which devices are polled). The autoconfiguration process assumes that `ioconf.c` exists and will complete the initialization of its structures by calling *xx*`probe()`, *xx*`attach()`, and *xx*`slave()`. See the *Overall Kernel Context* chapter for more information.
- *xx*.h — a set of header files, one for each driver. These header files define macros (e.g. `#define NSK 2`) that tell the drivers how many devices they will be managing. The drivers will use these macros *at compile time* to control conditional compilation and to size device tables.
- *mbglue.s* — contains assembly-level code that translates from the hardware interrupt mechanisms to the device-interrupt routines for the installed devices. *It does not exist on Sun-4 or Sun386i machines.*
- *Makefile* — a makefile that, when executed, will actually make the new kernel, compiling and linking files as necessary. Note that the entries in `/usr/sys/sun[234]/conf/files` (or `/usr/sys/sun386/conf/files`) refer to source files (i.e. `sundev/sk.c`), but that if `config` fails to find a named source file it will set up to use the corresponding object file (from the OBJ subdirectory of the configuration directory) instead. Thus, `config` works on both source licensed and object licensed machines.

7.2. An Example

The example that follows assumes that you're adding a driver for the Skeleton board (`sk.c`) to your system. To proceed, you will need a configuration directory and a config file for your new kernel. `config` will create a configuration directory in `/usr/sys/` with the same name as the new config file in `/usr/sys/conf`, so all you have to do is create that file:

```
example#  cd /usr/sys/sun[234]/conf
example#  cp GENERIC SKELETON
```

Then edit the `SKELETON` config file to reflect the presence, in your system, of the Skeleton board. As you can see by checking `config(8)`, each line in the file describes a different device — thus, you will simply need to add lines that describe the installation of the Skeleton board. The exact format of those lines will depend upon the address space within which the board is to be installed.

The address space that's given in the kernel config file will determine the address-space mappings that are set up by the MMU — the virtual addresses that the driver receives from the kernel, and then treats as pointers to the device's registers, will be within the address space given here. What's important is that the driver writer know and specify, at this point, the number of bits in the device address, and the number of bits in its data-access length.

The Skeleton board, as we've defined it, is an I/O-mapped Multibus device with an eight-bit status and an eight-bit data register. Thus, in a Sun-2 Multibus machine, it would be installed in I/O space; if we put it at offset `0x600` within that space, we'd add the following line to `SKELETON`:

```
device sk0 at mbio ? csr 0x600 priority 2
```

This says that we have an `sk` device (the first device is always, by convention, number 0) on the Multibus. The device has its control/status register (device register) at Multibus I/O address 0x600 (this is passed to *xx*`probe()` at boot time) and interrupts at level 2.

If our machine is a VMEbus machine, we will install the Skeleton device within `vme16d16` by way of a Multibus-VMEbus adapter. We choose `vme16d16` because it's the smallest address space:

```
device sk0 at vme16d16 ? csr 0x600 priority 2 vector skintr 0xC8
```

This says that, when plugged into an adapter board, the vector number `0xC8` is set up to route to the `skintr` routine. (Vector numbers `0xC8` through `0xFF` are reserved for user devices). Notice that `0x600` within `mbio` maps directly to `0x600` within `vme16d16`.

On a Sun-3 or Sun-4, it would likewise be reasonable to choose the smallest of the available address spaces:

Each of these config-file entries specify the installation of the Skeleton device for either a Multibus or a VMEbus system. It's fine for one config file to contain both entries — `config` will know the type of system that it is running on, and automatically use the right entry.

Only very rudimentary error checking is done on the config file. For example, if you declare a device attached to a controller, you must declare the controller as well. Also, a sanity check is done on the timezone and date entries. The checking, however, is not comprehensive.

One more point about the config file. The number of installed devices will be determined, for each driver, by `config`, and it will generate the appropriate `sk.h` header file for you in the configuration directory.

Now, you can go on with the process of building the new kernel. The next step is to edit `conf.c`, adding to it the names of the entry point routines for the Skeleton driver, and then installing those routines into the kernel's character device switch `cdevsw`. The following code accomplishes these two goals:

```
#include "sk.h"
#if NSK > 0
int skopen(), skclose(), skread(), skwrite(), skmmap();
#else
#define skopen  nodev
#define skclose nodev
#define skread  nodev
#define skwrite nodev
#define skmmap  nodev
#endif
    .
    .
    .
struct cdevsw cdevsw[] =
{
    .
    .
    {
skopen, skclose, skread, skwrite,
nodev, nodev, nodev, 0,
seltrue, skmmap,
    },
    .
    .
}
```

This will add the driver's routines to `cdevsw` if `NSK` is greater than 0 (`NSK` is, as already explained, calculated by `config`). Note well that the position in the `cdevsw` where we've installed our routines (the exact position depends, of course, upon how many device are already installed) is the same as the major device number which we will later assign to all devices driven by this driver — the major number is an index into `cdevsw`.

The entries in `cdevsw` are, in order, *xx*`open()`, *xx*`close()`, *xx*`read()`, *xx*`write()`, *xx*`ioctl()`, *xx*`stop()` and *xx*`reset()`, a `tty` structure pointer, and finally, *xx*`select()` and *xx*`mmap()`. The Skeleton driver doesn't have an *xx*`ioctl()` routine so this entry is set to `nodev`, the special routine that always returns an error. Since our device is not a tty it doesn't have an *xx*`stop()` routine (used for flow control) nor does it have a `tty` structure. *xx*`reset()` is *never* used so all devices set its entry to `nodev`. *xx*`select()` is called when a user process does a `select(2)` system call; it returns 1 if the device can be immediately selected. Since the Skeleton device is write only and arbitrarily fast, it's always selectable — so we will use the default `seltrue` routine that always returns 1.

The next step is to edit the file that tells `config` how to locate the driver source code. This source code will *not* be common to all Sun systems, and thus its pathname will go not into `/usr/conf.common/files_cmn` but into `/usr/sys/sun[234]/conf/files` (or `/usr/sys/sun386/conf/files`). Assuming that the driver source is in `/usr/sys/sundev`, here's the line you must add to `/usr/sys/sun[234]/conf/files` (or `/usr/sys/sun386/conf/files`):

```
sundev/sk.c optional sk device-driver
```

This says that the file `sundev/sk.c` contains the source code for the optional *sk* device and that it is a device driver.

After adding these lines to your configuration file, you can run `config`:

```
example# config SKELETON
```

`config` uses SKELETON, `/usr/conf.common/files_cmn` and `/usr/sys/sun[234]/conf/files` (or `/usr/sys/sun386/conf/files`) as input, and generates a number of files in the `../SKELETON` directory. One of these files is the `makefile` that contains a dependency tree for any new C source files you created during the process of adding new drivers (or whatever) to the kernel. `make` will use this as its command file when it is actually executed to produce the new kernel. When `config` finishes generating the `makefile`, it automatically goes on to generate the dependencies (unless you tell it not to with the **-n** command-line flag). The generation of the dependencies takes a long time, and before it starts, `config` will notify you with the message:

```
Doing a "make depend"
```

Now you can change directory to the new configuration directory, `../SKELETON` in this case, and make the new system:

```
example# cd ../SKELETON
example# make
```

Now you must add a new device entry to the `/dev` directory. The connections between the kernel and the device driver are established through the entries in the `/dev` directory. Using the example above as our model, we want to install the device for the Skeleton driver.

Device entries are made with one of two shell scripts in the `/dev` directory. The first, `MAKEDEV`, is for standard system devices and should be left as is. The second, `MAKEDEV.local`, differs only in that it contains entries for user devices, and it is here that entries for new devices should be placed.

It's worth looking inside `MAKEDEV` to see the kinds of things it does. The lines of shell script below reflect what you'd add to `MAKEDEV.local` for the new Skeleton device. First, there are some lines of commentary:

```
#! /bin/sh
#   MAKEDEV.local    4.45    86/04/15
# Graphics
#   sk* Skeleton Board
```

Then there's the actual shell code that makes the device entries:

```
sk*)
    unit=`expr $i : 'sk)'`
    /etc/mknod sk$unit c 40 $unit
    chmod 222 sk$unit
    ;;
```

This code extracts the numeric portion of `MAKEDEV.local`'s argument and passes it on to `mknod` and `chmod`. In the simplest case, we simply say:

```
example#  MAKEDEV.local sk0
```

`MAKEDEV.local` then makes the special inode `/dev/sk0` for a character special device with major device number 40 and minor device number 0, and then sets the mode of the file so that anyone can write to the device.

Having added the new device entry, you can install the new system and try it out.

```
example#  cp /usr/sys/sun[234]/SKELETON/vmunix vmunix+
example#  halt
          The system here goes through the halt sequence, then
          the monitor displays its prompt, at which point you can
          boot the system in single-user state
>   b vmunix+ -s
          The system boots up in single user state and
          then you can try things out
example#
```

If the system appears to work, save the old kernel under a different name and install the new one in /vmunix:

```
example#  cd /
example#  mv vmunix vmunix-
example#  mv vmunix+ vmunix
example#
```

Make sure that the new version of the kernel is actually called `vmunix` because programs like `ps` and `netstat()` use that exact name in collecting information they need from runtime tables. If the running version of the kernel is called something other than `vmunix` the results from such programs will be wrong.

7.3. Devices that use Two Address Spaces

Normally, devices interface to the system by way of a single address space. However, there are exceptions. Some Multibus devices have registers in Multibus I/O space *and* memory in Multibus memory space. And there are any number of VMEbus devices coming on the market that have memory in 24 or 32-bit VME space while keeping their control and status registers in 16, or even 8-bit, VME space.

Unfortunately, such situations can't currently be handled in a clean fashion because the kernel configuration program (`config`) can't cope with dual-space devices. The *xx*`probe()` routine is the core of the problem, since it deals with only a single space.

There are, fortunately, two ways to work around the problem:

- The first is easier, but rather inelegant. It consists of treating the device as if it were two devices, and of writing two separate "drivers" for it. So, for example, if we were to have a new, dual-space, VMEbus version of the Skeleton device, we'd add the following *two* lines to the config file:

```
# Skeleton Memory Space
device skm0 at vme32d32 ? csr 0xD0000000 priority 3
# Skeleton Register Space
device skr0 at vme16d16 ? csr 0xD000 priority 3 vector skintr 0x88
```

  It's also necessary to have two entries in `/usr/sys/sun[234]/conf/files` (or `/usr/sys/sun386/conf/files`):

```
sundev/skm.c        optional skm device-driver
sundev/skr.c        optional skr device-driver
```

  And it's necessary to have a second "driver". Actually, all of the real driver code goes into `skr.c`, which manipulates the device registers. The second driver, `skm.c`, consists entirely of a `probe()` routine — all its other routines are null.

  Both sides of the driver, `skr.c` and `skm.c`, include the same register header file `skreg.h`. `skreg.h` contains an *external* declaration for an array of structures (one for each instance of the device) that contain whatever information `skr.c` needs from the memory-side `probe()` routine:

```
extern struct sk_devinfo sk_devinfo[NSK];
```

  All that remains is for the memory-side `probe()` routine to initialize `sk_devinfo`.

- There's a second procedure for installing dual-space devices. It's a bit harder to use, but it doesn't require a stub driver containing only a `probe()` routine.

Pick one of the two device installation addresses for normal treatment in the config file. It doesn't matter which one you pick, unless the device is a memory-mapped Multibus device, in which case you must pick the address in Multibus Memory space. Otherwise just pick the one that's most convenient for your *xx*`probe()` routine to use to test the device installation. The registers and memory in this first space will then be automatically mapped into kernel virtual space (as usual) by the autoconfiguration process.

Then use the config file `flags` word to communicate the second space installation address to your driver. The driver will then find that address in `md->md_flags` and be able to access it from either the *xx*`attach()` or *xx*`slave()` routine; it's best (for most character devices) to pick it up at *xx*`attach()` time. The driver can then use `rmalloc()` to allocate (from `kernelmap`) virtual space for the second-space registers/memory, and then call `mapin()` to map them into kernel space. (See the *Kernel Support Routines* appendix for details about `mapin()`).

7.4. Adding and Removing Loadable Drivers

The Sun386i supports loadable drivers. A loadable driver doesn't need to be linked with the kernel `.o` files. Nor does the system have to be rebooted or rebuilt for loadable drivers to be used. You can simply add a loadable driver to a running system. Once you have a driver in the loadable form, you can load it into the running system with the `modload`(8) command. You must be the superuser to do this.

Take care when loading an undebugged driver for the first time. Although there are many consistency checks made when a driver is loaded, it is still possible for drivers to crash the system. One of the more common crashes occurs when the running kernel is not `/vmunix`. `modload` assumes by default (unless the `-A` switch is provided) that the running kernel is `/vmunix`. It resolves driver references to kernel addresses by reading the symbol table from `/vmunix`. If `/vmunix` is not the running system, then the system is likely to crash when the driver is used.

A typical example of the `modload` command is:

```
example# modload zz.o -conf <config_file> -exec <exec_file>
```

This tells the kernel that the driver object module is in `zz.o`. (See the *Loadable Drivers* section of the *Driver Development Topics* chapter for information about how to build a loadable driver.)

Configuration information for the driver and optionally the block and character major numbers are specified in the file *config_file*. If `modload` is successful, the file *exec_file* is executed. This file is typically a script used to make the `/dev` entries for the driver. `modload`(8) has many options; see its man page for details.

Error messages from `modload` can appear in two places. The `modload` utility itself prints error messages to standard output on the terminal from which `modload` is run. In addition, modload-related kernel code can print information to the console. For this reason, we recommend that the console output be visible when you issue the `modload` command.

When it is loading a driver, `modload` may fail for a variety of reasons. For example, the driver initialization routine may not do all that is required (as described in the *Loadable Drivers* section of the *Driver Development Topics* chapter). Or the linkage structure in the driver wrapper module may have invalid addresses. Since it is not possible to return a unique error code for every possible condition, a single error code is returned and additional information is often printed on the console. To assist the driver writer in debugging the driver, the kernel variable `vddebug` can be set to `-1` using `adb` or `kadb`. This will cause the kernel to print additional informational messages when loading a module.

To inquire about device drivers after they are loaded, use the `modstat(8)` command. It displays the module id of the driver, the name of the device, and the major numbers of the block and character devices, as well as some additional information about the module.

The module id is required to unload a driver. A driver can be unloaded by using the `modunload(8)` command, as in this example:

```
example# modunload -id 2 -exec <exec_file>
```

This example assumes that the `modstat` command displayed the driver's module id as 2. The file *exec_file* is executed and if the execution is successful the driver is unloaded. Typically this file is a script that removes the `/dev` entries for the driver.

An example of a script that could be used with `modload` is as follows:

```
#!/bin/csh -f
if $3 != "0" then
    if ( ! -r /dev/zz0) then
        echo /etc/mknod /dev/zz0 b $3 0
        /etc/mknod /dev/zz0 b $3 0
    endif
endif

if $4 != "0" then
    if ( ! -r /dev/xrfd0a) then
        echo /etc/mknod /dev/xrfd0a c $4 0
        /etc/mknod /dev/rzz0 c $4 0
    endif
endif
```

The script is invoked with the following arguments:

```
<module_id> <module_type> <block_major_number> <character_major_number>
```

`modunload` could be invoked with the following script to remove the `/dev` entries for the driver:

```
#!/bin/csh -f
rm -f /dev/zz0
rm -f /dev/rzz0
```

8

Pseudo-Device Drivers — A Ramdisk

Pseudo-Device Drivers — A Ramdisk **151**

8.1. A Ramdisk Driver 152
  Ramdisk Source Code 152
  Ramdisk Installation 153
  Ramdisk Test Program 156

8

Pseudo-Device Drivers — A Ramdisk

SunOS supports "software devices", sometimes called "pseudo devices", which have no associated physical devices. Such devices can be quite useful. The system memory devices, for example, are pseudo devices, and they can be used to access installed peripheral devices, as is shown in the discussion of frame-buffer installation in *Direct Opening of Memory Devices* section of this manual. The memory devices allow such direct physical-device access by providing a means by which processes can read and write physical memory outside their own address space. For example, the `ps` command uses the `kmem` pseudo-device driver to access the kernel's process tables by way of the physical memory to which the kernel is mapped.

This section will introduce pseudo-devices by way of a real, working pseudo-device ramdisk. As you will see, such a ramdisk requires none of the subtlety that makes physical disk drivers so difficult.[6] Yet it does buy speed, since ramdisks avoid two distinct kinds of file-system overhead:

- In normal use, IO buffers get paged out, despite the use of the kernel buffer cache to minimize unnecessary I/O operations. A ramdisk is an especially big win on reads, since reading processes must normally block while requested data is brought into the buffer cache.
- During normal file-system operation, file control information (like inodes) must be written synchronously with data. This overhead doesn't exist with ramdisks.

Ramdisks can be used for `/tmp`. This way, if a system crash results in the loss of ramdisk files, it's not a serious problem. Note that for some applications, particularly those that involve temporary files larger than ramdisk memory, using `/tmp` isn't a very good idea. An alternative is to mount the ramdisk as `/aux`, and to use it explicitly each time you think it's safe. Ramdisks have only a minimal impact on applications software — once they're set up they are entirely transparent. (Note that ramdisks — like devices in general — can be shared by multiple processes. This driver can thus be used as an indirect means of sharing memory.)

---

[6] The ramdisk given here is *very* crude. A production version should have its memory allocated at boot time and should be pagable. And with the memory-management system introduced in SunOS 4.0, a ramdisk probably won't improve performance anyway. In general, you'll be better off letting UNIX manage memory as a page cache, rather than devoting some of that cache to a ramdisk

8.1. A Ramdisk Driver

The following ramdisk driver consumes a half-megabyte of kernel memory, which is allocated to the ramdisk pseudo-device.

Ramdisk Source Code

Put the source code for the ramdisk driver into `/sys/sundev/ram.c`.

```
/*
 * Ramdisk pseudo-device to support I/O to real memory
 * (a statically allocated kernel array).
 */

#include "ram.h"
#if NRAM > 0
#include "../h/param.h"     /* Includes "../h/types.h" */
#include "../h/errno.h"
#include "../h/uio.h"
#include "../h/buf.h"

#define RAMSIZE (1024*512)    /* Half a megabyte */
char ram[NRAM][RAMSIZE];

ramopen(dev,wrtflag)
    dev_t dev;
    int wrtflag;
{
    return(minor(dev) >= NRAM ? ENXIO : 0);
}

ramsize(dev)
    dev_t dev;
{
    return(minor(dev) >= NRAM ? -1 : btodb(RAMSIZE));
}

ramread(dev,uio)
    dev_t dev;
    register struct uio *uio;
{
    if ((unsigned)uio->uio_offset > RAMSIZE)
        return(EINVAL);
    return(uiomove(ram[minor(dev)]+uio->uio_offset,
        MIN(uio->uio_resid, RAMSIZE - uio->uio_offset),
        UIO_READ, uio));
}

ramwrite(dev,uio)
    dev_t dev;
    register struct uio *uio;
{
    if ((unsigned)uio->uio_offset > RAMSIZE)
        return(EINVAL);
    return(uiomove(ram[minor(dev)]+uio->uio_offset,
        MIN(uio->uio_resid, RAMSIZE - uio->uio_offset),
        UIO_WRITE, uio));
}

ramstrategy(bp)
    register struct buf *bp;
{
    register long offset = dbtob(bp->b_blkno);

    if ((u_long)offset > RAMSIZE) {
        bp->b_error = EINVAL;
        bp->b_flags |= B_ERROR;
    } else {
        caddr_t raddr = ram[minor(bp->b_dev)]+offset;
        unsigned nbytes = MIN(bp->b_bcount, RAMSIZE-offset);

        if (bp->b_flags&B_READ)
            bcopy(raddr, bp->b_un.b_addr, nbytes);
        else
            bcopy(bp->b_un.b_addr, raddr, nbytes);
        bp->b_resid = bp->b_bcount - nbytes;
    }
    iodone(bp);
}
#endif
```

Pseudo-device drivers, by definition, have no corresponding physical devices. Thus, they have no probe routines.

Note the routine `ramsize`. All block drivers provide such a routine, which is charged with returning the sector size of the device in the peculiar units which the kernel expects. (This information is then used to maximize the speed of `fsck`). `ramsize()` calls the `btodb()`, conversion routine, passing it an argument in bytes, and receiving from it an appropriately scaled result.

Ramdisk Installation

The more detailed discussion of these and related configuration procedures can be found in the *Configuring the Kernel* chapter of this manual.

First, create the file `/sys/sundev/ram.h` containing the line:

```
#define NRAM 1
```

Then, edit `/usr/sys/sun[234]/conf/files` or `/usr/sys/sun386/conf/files`, adding the following line to the end of it:

```
sundev/ram.c            optional ram device-driver
```

Then, edit both the `bdevsw` and `cdevsw` arrays in `/sys/sun/conf.c`, adding entries for the ramdisk to each of them. (In this discussion, we will only use the ramdisk as a block device, but the driver provides all the entry points necessary for use as either a block or a character driver).

```
#include "ram.h"
if NRAM > 0
int ramopen(), ramread(), ramwrite();
int ramstrategy(), ramsize();
#else
#define ramopen       nodev
#define ramread       nodev
#define ramwrite      nodev
#define ramstrategy   nodev
#define ramsize       nodev
#endif
    .
    .
    .
    {
ramopen, nulldev, ramstrategy, nulldev, /* 8 */
ramsize, nulldev
    }
    .
    .
    .
    {
ramopen, nulldev, ramread, ramwrite,    /* 30 */
nodev, nodev, nulldev, 0, seltrue, nodev,
    }
```

Next, move into `/dev` and create device entries to correspond to the entries in `conf.c`.

```
example# cd /dev
example# /etc/mknod ram0c 8 0
example# /etc/mknod rram0c 30 0
```

The next step is to make a new configuration directory for the variant of you kernel that will include the ramdisk. Copy your kernel configuration file and add the line:

```
pseudo-device       ram
```

to the pseudo-device section of the copy. If your config file was named `GENERIC`, you might name the ramdisk variation `GENERIC_RAM`.

Then, make a version of the system kernel that includes the ramdisk:

```
example# mkdir /sys/GENERIC_RAM
example# /etc/config GENERIC_RAM
example# cd ../GENERIC_RAM
example# make depend
example# make
example# cp /vmunix /vmunix.old
example# cp vmunix /vmunix
example# /etc/reboot
```

During the reboot, note that the size of the kernel has gotten very large. After the reboot, make and associate a ``filesystem'' with the block ramdisk device:

```
/etc/mkfs /dev/ram0c 1024 8 8 4096 1024 16 5 100
/etc/mount /dev/ram0c /tmp
```

That's 1024 blocks total (512 Kb), broken out as 8 sectors of 8 tracks of 4096 bytes per block with 1024 byte fragment size with 16 cylinders per group with 5% minimum free (as in `df(1)`) and 100 revolutions per second. (This two line sequence should probably be put in the `/etc/rc.local` script).

Once the ramdisk filesystem is mounted onto `/tmp`, then any program which creates and uses files on `/tmp` will use the ramdisk. Reads and writes to these files will be very fast. Measured performance indicates that I/O on files of about 10K bytes is about 5 times as fast as with a physical disk, and that this factor increases to about 10 for very large files.

Ramdisk Test Program

Here's a test program that proves that the ramdisk works:

```
#define BUFSIZ 1024
#define CYCLES 100
#define RAMDISK

/*
 * Ramdisk test program
 */

main()
{
   int fd;                    /* file descriptor */
   int nb;                    /* number of bytes transferred */
   int i;                     /* generic loop counter variable */
   int count=BUFSIZ;
   char buffer[BUFSIZ];
   int iterations=0, error=0, done=0;

#ifndef RAMDISK
   /* Open a file on the regular filesystem */
   if ((fd = open("testfile", 2)) == -1 ) {
           perror("ramdisk test (normal opening)");
           exit(1);
   }
#else
   /* Open a file in the ram disk filesystem */
   if ((fd = open("/tmp/testfile", 2)) == -1 ) {
           perror("ramdisk test (ram opening)");
           exit(1);
   }
#endif

   do {
           lseek(fd, 0, 0);
           if (write(fd, buffer, count) != count) {
                   perror("ramdisk test (writing)");
                   exit(1);
           }
           lseek(fd, 0, 0);
           if (read(fd, buffer, count) != count) {
                   printf("count= %d0, count);
                   perror("ramdisk test (reading)");
                   exit(1);
           }
           if (iterations++ == CYCLES ) done++;
   }
   while ( !error && !done );
   close(fd);
   exit(0);
}
```

PART TWO: STREAMS Programming

9

Introduction to STREAMS

Introduction to STREAMS **161**

9.1. A Basic View of a Stream 162
  System Calls 163
9.2. Benefits of STREAMS 165
  Creating Service Interfaces 165
  Manipulating Modules 165
  Protocol Portability 165
  Protocol Substitution 166
  Protocol Migration 166
  Module Reusability 167
9.3. An Advanced View of a Stream 168
  Stream Head 169
  Modules 169
  Stream End 170
9.4. Building a Stream 171
  Expanded Streams 172
  Pushable Modules 172
9.5. Basic User Level Functions 173
  STREAMS System Calls 173
  An Asynchronous Protocol Stream Example 174
    Initializing the Stream 175
    Message Types 176
    Sending and Receiving Messages 176

Using Messages in the Example 177
Other User Functions 180
9.6. Kernel Level Functions 180
Messages 180
Message Allocation 182
Put and Service Procedures 183
Put Procedures 183
Service Procedures 183
Kernel Processing 184
Read Side Processing 185
Driver Processing 185
CHARPROC 185
CANONPROC 186
Write Side Processing 186
Analysis 187
9.7. Other Facilities 187
Message Queue Priority 187
Flow Control 188
Multiplexing 190
Monitoring 192
Error and Trace Logging 193
9.8. Driver Design Comparisons 195
Environment 195
Drivers 195
Modules 196
9.9. Glossary 196

9

Introduction to STREAMS

STREAMS were designed to systematize the existing UNIX character I/O mechanism and to support the development of communications services.

STREAMS consist of a set of system calls, kernel resources and kernel routines. For detailed information about the STREAMS-kernel interface, about the internal structure of STREAMS modules and about STREAMS driver programming, see the following chapters.

The UNIX system was originally designed as a general-purpose, multi-user, interactive operating system for minicomputers. Initially developed in the 1970's, the system's communications environment included slow to medium speed, asynchronous terminal devices. The original design, the communications environment, and hardware state of the art influenced the character I/O mechanism but the character I/O area did not require the same emphasis on modularity and performance as other areas of the system.

Support for a broader range of devices, speeds, modes, and protocols has since been incorporated into the system, but the original character I/O mechanism, which processes one character at a time, made such development difficult. Additionally, a paucity of tools and the absence of a framework for incorporating contemporary networking protocols added to the difficulty.

Attempts to compensate for the above problems have led to diverse, ad-hoc implementations; for example, protocol drivers are often intertwined with the hardware configuration in which they were developed. As a result, functionally equivalent protocol software often cannot interface with alternate implementations of adjacent protocol layers. Portability, adaptability, and reuse of software have been hindered.

STREAMS, a general, flexible facility and a set of tools for development of UNIX system communication services, is intended to remedy these problems. STREAMS supports services ranging from complete networking protocol suites to individual device drivers.

STREAMS defines standard interfaces for character I/O within the kernel, and between the kernel and the rest of the system. The associated mechanism is simple and open-ended. It consists of a set of system calls, kernel resources, and kernel utility routines. The standard interface and open-ended mechanism enable modular, portable development and easy integration of higher performance network services and their components. STREAMS does not impose any specific network architecture. Instead, it provides a powerful framework with a consistent user interface that is compatible with the existing character I/O interface—which is still available.

STREAMS modularity and design reflect the "layers and options" characteristics of contemporary networking architectures. The basic components in a STREAMS implementation are referred to as modules. These modules, which reside in the kernel, offer a set of processing functions and associated service interfaces. From user level, modules can be dynamically selected and interconnected to provide any rational processing sequence. Kernel programming, assembly, and link editing are not required to create the interconnection. Modules can also be dynamically "plugged into" existing connections from user level. STREAMS modularity allows:

- User level programs that are independent of underlying protocols and physical communication media.
- Network architectures and higher level protocols that are independent of underlying protocols, drivers, and physical communication media.
- Higher level services that can be created by selecting and connecting lower level services and protocols.
- Enhanced portability of protocol modules resulting from STREAMS' well-defined structure and interface standards.

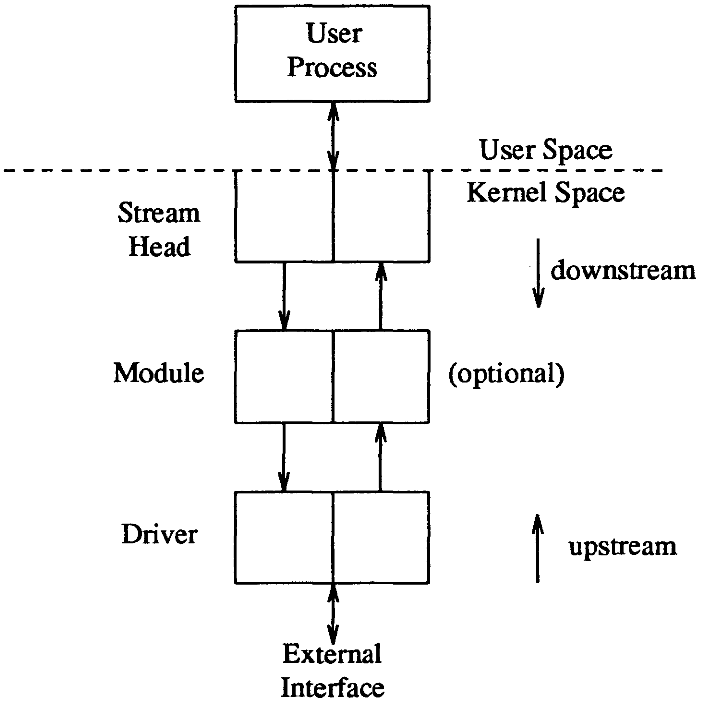
In addition to modularity, STREAMS provides developers with integral functions, a library of utility routines, and facilities that expedite software design and implementation. The principal facilities are:

- Buffer management – To maintain STREAMS' own, independent buffer pool.
- Flow control – To conserve STREAMS' memory and processing resources.
- Scheduling – To incorporate STREAMS' own scheduling mechanism.
- Multiplexing – For processing interleaved data streams, such as occur in SNA, X.25, and windows.
- Asynchronous operation of STREAMS and user processes – Allows STREAMS-related operations to be performed efficiently from user level.
- Error and trace loggers – For debugging and administrative functions.

9.1. A Basic View of a Stream

"STREAMS" is a collection of system calls, kernel resources, and kernel utility routines that can create, use, and dismantle a "Stream". A Stream is a full-duplex processing and data transfer path between a driver in kernel space and a process in user space (see Figure *9-1*).

Figure 9-1 *Basic Stream*



A Stream has three parts: A Stream head, module(s) (optional), and a driver (also referred to as the Stream end). The Stream head provides the interface between the Stream and user processes. Its principal function is to process STREAMS-related user system calls. A module processes data that travel between the Stream head and driver. A STREAMS driver may be a device driver, providing the services of an external I/O device, or an internal software driver, commonly called a pseudo-device driver.

Using a combination of system calls, kernel routines, and kernel utilities, STREAMS passes data between a driver and the Stream head in the form of messages. Messages that are passed from the Stream head toward the driver are said to travel downstream, and messages passed in the other direction travel upstream.

The Stream head transfers data between the data space of a user process and STREAMS kernel data space. Data sent to a driver from a user process are packaged into STREAMS messages and passed downstream. Messages arriving at the Stream head from downstream are processed by the Stream head, and data are copied into user buffers. STREAMS can insert one or more modules into a Stream between the Stream head and driver to perform intermediate processing of data passing between the Stream head and driver.

System Calls

Applications programmers can use the STREAMS facilities via a set of system calls. This system call interface is upward compatible with the existing character I/O facilities. The `open(2)` system call will recognize a STREAMS file and create a Stream to the specified driver. A user process can send and receive data using `read(2)` and `write(2)` in the same manner as with character files and devices. The `ioctl(2)` system call enables application programs to perform functions specific to a particular device. In addition, a set of generic STREAMS `ioctl()` commands (see `streamio`(4)) support a variety of functions for accessing and controlling Streams. A `close(2)` will dismantle a Stream.

`open()`, `close()`, `read()`, `write()`, and `ioctl()` support the basic set of operations on Streams. In addition, new system calls support advanced STREAMS facilities. The `poll(2)` system call enables an application program to poll multiple Streams for various events. When used with the STREAMS `I_SETSIG` `ioctl()` command, `poll()` allows an application to process I/O in an asynchronous manner. The `putmsg(2)` and `getmsg(2)` system calls enable application programs to interact with STREAMS modules and drivers through a service interface (described next).

These calls are discussed in this chapter and, in more detail, the the chapters that follow. They are precisely specified in the following manual pages:

Figure 9-2 *STREAMS-Related Manual Pages*

Man Page	*Description*
`open.2`	Open a stream
`close.2`	Close a stream
`read.2`	Read from a stream
`write.2`	Write to a stream
`putmsg.2`	Send a message on a stream
`getmsg.2`	Get next message off a stream
`poll.2`	STREAMS input/output multiplexing
`clone.4`	Open any minor device on a STREAMS driver
`streamio.4`	STREAMS ioctl commands
`termio.4`	General terminal interface
`tty_compat.4m`	V7/4BSD compatibility STREAMS module
`tty_std.4m`	Standard terminal STREAMS module
`kbd.4s`	Sun keyboard device
`kb.4m`	Sun keyboard STREAMS module
`mouse.4s`	Sun mouse device
`ms.4m`	Sun mouse STREAMS module
`pty.4`	Pseudo terminal driver
`console.4s`	Sun console driver and terminal emulator
`mti.4s`	Systech MTI-800/1600 multi-terminal interface
`zs.4s`	Zilog 8530 SCC serial communications drive
`nit.4m`	Network Interface Tap (NIT) Protocol
`nit_if.4m`	STREAMS NIT device interface
`nit_pf.4m`	STREAMS NIT packet filtering module
`nit_buf.4m`	STREAMS NIT buffering module

9.2. Benefits of STREAMS

STREAMS offers two major benefits for applications programmers: easy creation of modules that offer standard data communications services, and the ability to manipulate those modules on a Stream.

Creating Service Interfaces

One benefit of STREAMS is that it simplifies the creation of modules that present a service interface to any neighboring application program, module, or device driver. A service interface is defined at the boundary between two neighbors. In STREAMS, a service interface is a specified set of messages and the rules for allowable sequences of these messages across the boundary. A module that implements a service interface will receive a message from a neighbor and respond with an appropriate action (for example, send back a request to retransmit) based on the specific message received and the preceding sequence of messages.

STREAMS provides features that make it easier to design various application processes and modules to common service interfaces. If these modules are written to comply with industry-standard service interfaces, they are called protocol modules.

In general, any two modules can be connected anywhere in a Stream. However, rational sequences are generally constructed by connecting modules with compatible protocol service interfaces. For example, a module that implements an X.25 protocol layer, as shown in Figure *9-3*, presents a protocol service interface at its input and output sides. In this case, other modules should only be connected to the input and output side if they have the compatible X.25 service interface.

Manipulating Modules

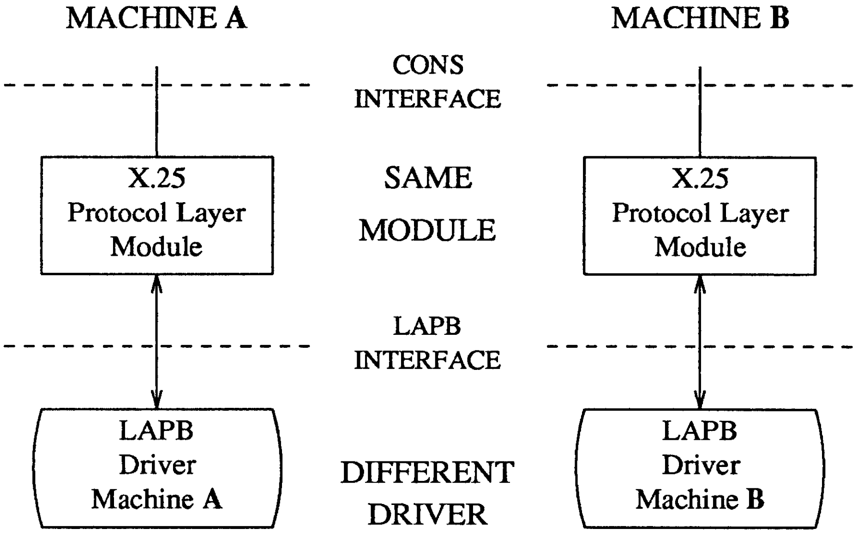
STREAMS provides the capabilities to manipulate modules from user level, to interchange modules with common service interfaces, and to present a service interface to a Stream user process. As mentioned above, these capabilities yield benefits when implementing networking services and protocols, including:

- User level programs can be independent of underlying protocols and physical communication media.
- Network architectures and higher level protocols can be independent of underlying protocols, drivers and physical communication media.
- Higher level services can be created by selecting and connecting lower level services and protocols. Below are examples of the benefits of STREAMS capabilities to developers for creating service interfaces and manipulating modules.

Protocol Portability

Figure *9-3* shows how the same X.25 protocol module can be used with different drivers on different machines by implementing compatible service interfaces. The X.25 protocol module interfaces are Connection Oriented Network Service (CONS) and Link Access Protocol – Balanced (LAPB) driver.

Figure 9-3 *Protocol Module Portability*


MACHINE A
MACHINE B
CONS
INTERFACE
X.25
Protocol Layer
Module
SAME
MODULE
X.25
Protocol Layer
Module
LAPB
INTERFACE
LAPB
Driver
Machine A
DIFFERENT
DRIVER
LAPB
Driver
Machine B

Protocol Substitution

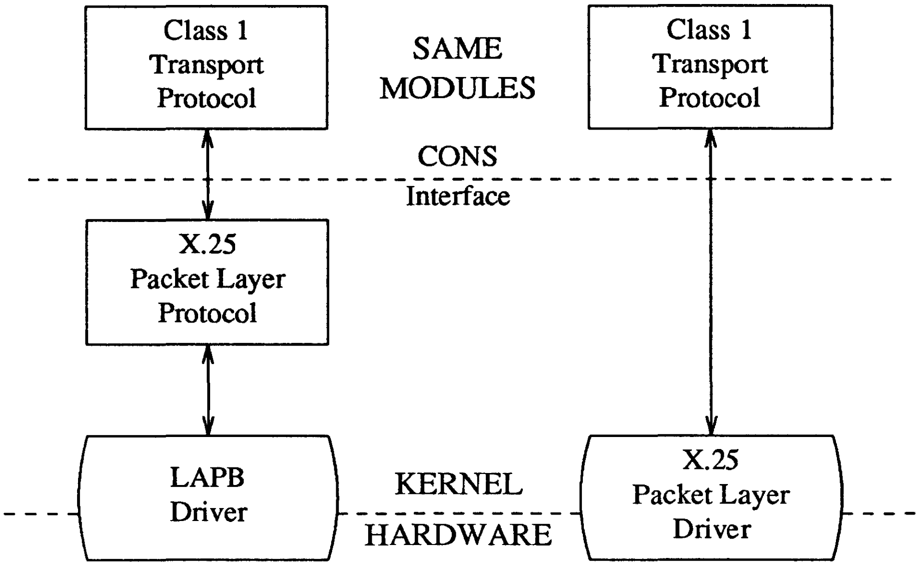
Alternative protocol modules (and device drivers) can be interchanged on the same machine if they are implemented to an equivalent service interface(s).

Protocol Migration

Figure *9-4* illustrates how STREAMS can migrate functions between kernel software and front end firmware. A common downstream service interface allows the transport protocol module to be independent of the number or type of modules below. The same transport module will connect without modification to either an X.25 module or X.25 driver that has the same service interface.

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

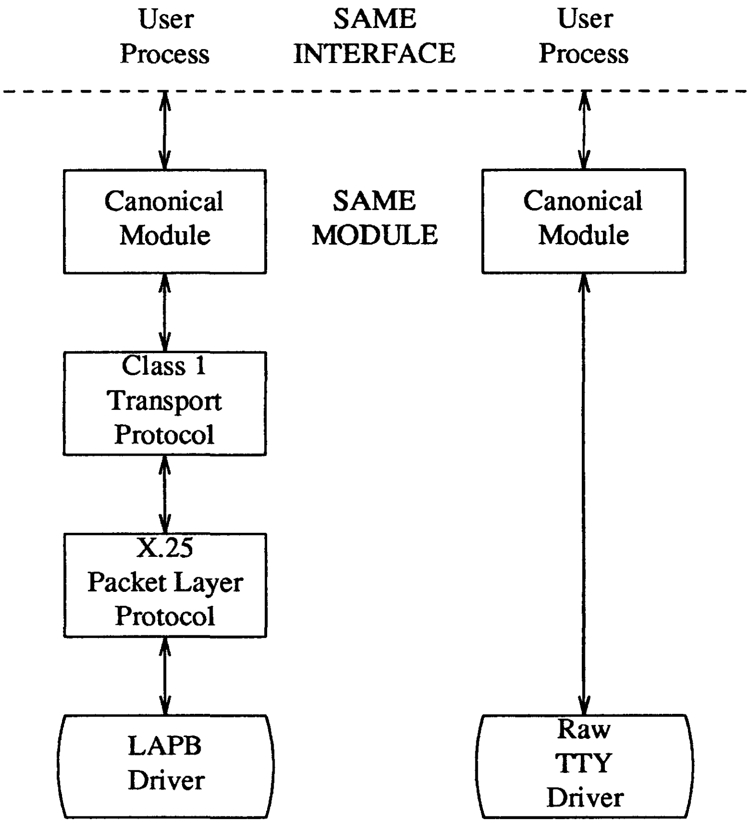
Figure 9-4 *Protocol Migration*


Class 1 Transport Protocol
SAME MODULES
Class 1 Transport Protocol
CONS
Interface
X.25 Packet Layer Protocol
LAPB Driver
KERNEL
HARDWARE
X.25 Packet Layer Driver

Module Reusability

Figure *9-5* 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.

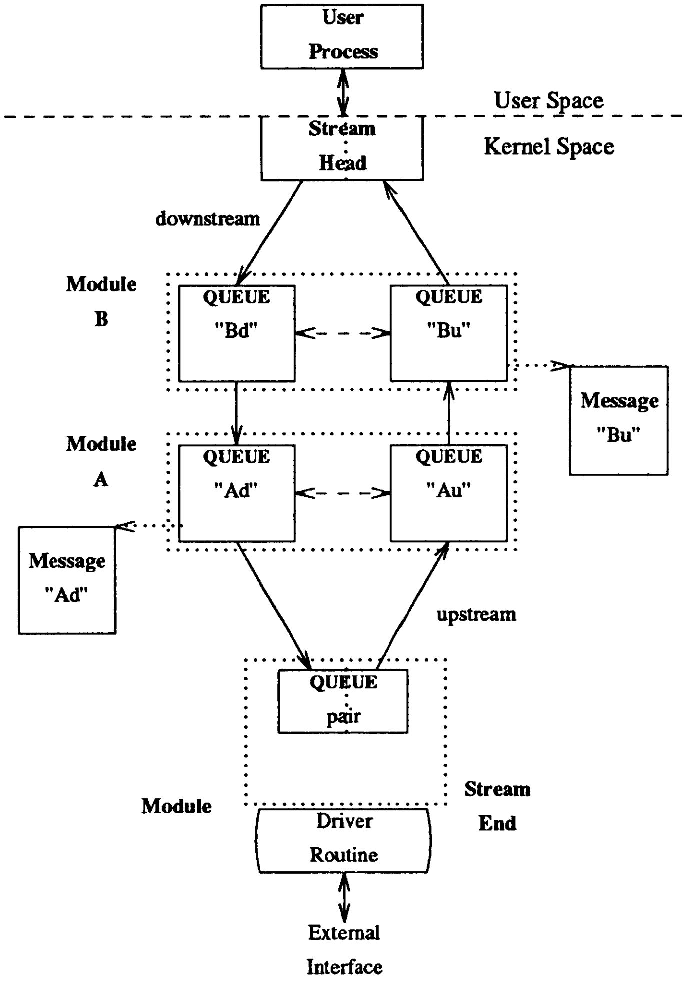
Figure 9-5 *Module Reusability*


User Process
SAME INTERFACE
User Process
Canonical Module
SAME MODULE
Canonical Module
Class 1 Transport Protocol
X.25 Packet Layer Protocol
LAPB Driver
Raw TTY Driver

9.3. An Advanced View of a Stream

The STREAMS mechanism constructs a Stream by serially connecting kernel resident STREAMS components, each constructed from a specific set of structures. As described earlier and shown in Figure *9-6*, the primary STREAMS components are the Stream head, optional module(s), and Stream end.

Figure 9-6 *Stream In More Detail*


User Process
User Space
Kernel Space
Stream Head
downstream
Module B
QUEUE "Bd"
QUEUE "Bu"
Message "Bu"
Module A
QUEUE "Ad"
QUEUE "Au"
Message "Ad"
upstream
QUEUE pair
Module
Stream End
Driver Routine
External Interface

Stream Head

The Stream head provides the interface between the Stream and an application program. The Stream head processes STREAMS-related system calls from the application and performs the bidirectional transfer of data and information between the application (in user space) and messages (in STREAMS' kernel space).

Messages are the only means of transferring data and communicating within a Stream. A STREAMS message contains data, status/control information, or a combination of the two. Each message includes a specified message type indicator that identifies the contents.

Modules

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

Each module is constructed from a pair of QUEUE structures (see Au/Ad and Bu/Bd in Figure *9-6*). A pair is required to implement the bidirectional and symmetrical attributes of a Stream. One QUEUE performs functions on messages passing upstream through the module (Au and Bu in Figure *9-6*). The other set (Ad and Bd) performs another set of functions on downstream messages. (A QUEUE, which is part of a module, is different from a message queue, which is described later.)

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

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

Each QUEUE in a module may contain or point to messages, processing procedures, or data:

- Messages – These are dynamically attached to the QUEUE on a linked list ("message queue", see Au and Bd in Figure *9-6*) as they pass through the module.
- Processing procedures – A put procedure, to process messages, must be incorporated in each QUEUE. An optional service procedure, to share the message processing with the put procedure, can also be incorporated. According to their function, the procedures can send messages upstream and/or downstream, and they can also modify the private data in their module.
- Data – Developers may provide private data if required by the QUEUE to perform message processing (for example, state information and translation tables).

In general, each of the two QUEUEs in a module has a distinct set of all of these elements. Additional module elements will be described later. Although depicted as distinct from modules (see Figure *9-6*), a Stream head and the Stream end also contain a pair of QUEUEs.

Stream End

A Stream end is a module in which the module's processing procedures are the driver routines. The procedures in the Stream end are different from those in other modules because they are accessible from an external device and because the STREAMS mechanism allows multiple Streams to be connected to the same driver.

The driver can be a device driver, providing an interface between kernel space and an external communications device, or an internal pseudo-device driver. A pseudo-device driver is not directly related to any external device, and it performs functions internal to the kernel. The multiplexing driver discussed in the *Other Facilities* chapter is a pseudo-device driver.

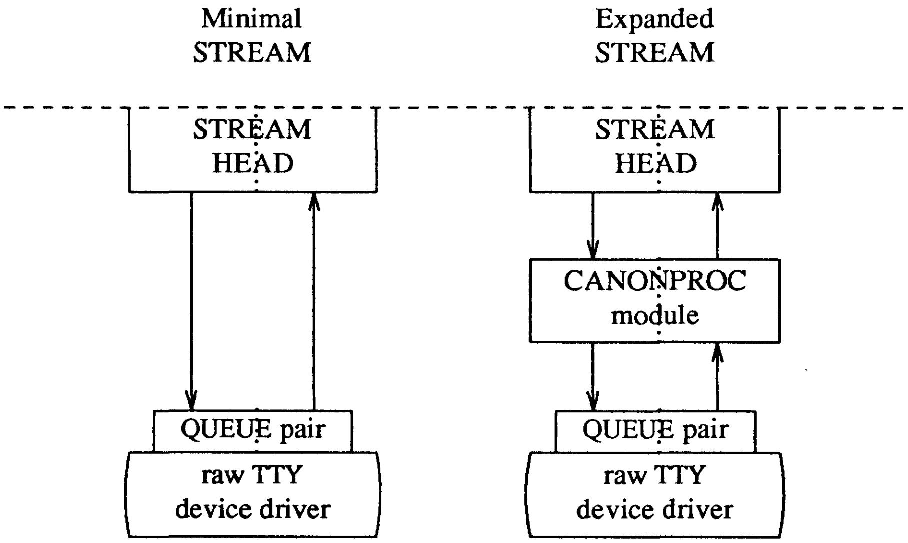
Device drivers must transform all data and status/control information between STREAMS message formats and their external representation. Differences between STREAMS and character device drivers are discussed in the *Driver Design Comparisons* chapter.

9.4. Building a Stream

A Stream is created on the first `open(2)` system call to a character special file corresponding to a STREAMS driver. A STREAMS device is distinguished from other character devices by a field contained in the associated `cdevsw` device table entry.

A Stream is usually built in two steps. Step one creates a minimal Stream consisting of just the Stream head and device driver, and step two adds modules to produce an expanded Stream (see Figure 9-7). The first step has three parts: head and driver structures are allocated and initialized; the modules in the head and end are linked to each other to form a Stream; the driver open routine is called.

Figure 9-7 *Setting Up a Stream*



If the driver performs all character and device processing required, no modules need be added to a Stream. Examples of STREAMS drivers include a raw tty driver (one that passes along input characters without change) and a driver with multiple Streams open to it (corresponding to multiple minor devices opened to a character device driver).

When the driver receives characters from the device, it places them into messages. The messages are then transferred to the next Stream component, the Stream head, which extracts the contents of the message and copies them to user space. Similar processing occurs for downstream character output; the Stream head copies data from user space into messages and sends them to the driver.

Expanded Streams

As the second step in building a Stream, modules can be added to the Stream. In the right-hand Stream in Figure 9-7, the CANONPROC module was added to provide additional processing on the characters sent between head and driver.

Modules are added and removed from a Stream in last-in-first-out (LIFO) order. They are inserted and deleted at the Stream head via the `ioctl(2)` system call. In the Stream on the left of Figure 9-5, the X.25 module was the first added to the Stream, followed by Class 1 Transport and Canonical modules. To replace the Class 1 module with a Class 0 module, the Canonical module would have to be removed first, then the Class 1 module, then a Class 0 module would be added and the Canonical module put back.

Because adding and removing modules resembles stack operations, the add is called a push and the remove a pop. Push and pop are two of the `ioctl()` functions included in the STREAMS subset of `ioctl()` system calls. These commands perform various manipulations and operations on Streams. The modules manipulated in this manner are called pushable modules, in contrast to the modules contained in the Stream head and end. This stack terminology applies only to the setup, modification, and breakdown of a Stream.

Subsequent use of the word "module" will refer to those pushable modules between Stream head and end.

The Stream head processes the `ioctl()` and executes the push, which is analogous to opening the Stream driver. Modules are referenced by a unique symbolic name, contained in the STREAMS `fmodsw` module table (similar to the `cdevsw` table associated with a device file). The module table and module name are internal to STREAMS and are accessible from user space only through STREAMS `ioctl()` system calls. The `fmodsw` table points to the module template in the kernel. When a module is pushed, the template is located, the module structures for both `QUEUE`s are allocated, and the template values are copied into the structures.

In addition to the module elements described in *A Basic View of a Stream*, each module contains pointers to an open routine and a close routine. The open is called when the module is pushed, and the close is called when the module is popped. Module open and close procedures are similar to a driver open and close.

As in other files, a STREAMS file is closed when the last process open to it closes the file by a `close(2)` system call. This system call causes the Stream to be dismantled (modules popped and the driver close executed).

Pushable Modules

Modules are pushed onto a Stream to provide special functions and/or additional protocol layers. In Figure 9-7, the Stream on the left is opened in a minimal configuration with a raw tty driver and no other module added. The driver receives one character at a time from the device, places the character in a message, and sends the message upstream. The Stream head receives the message, extracts the single character, and copies it into the reading process buffer to send to the user process in response to a `read(2)` system call. When the user process wants to send characters back to the driver, it issues a `write(2)` system call, and the characters are sent to the Stream head. The head copies the characters into one or more multi-character messages and sends them downstream. An application program requiring no further kernel character processing would use this minimal Stream.

A user requiring a more terminal-like interface would need to insert a module to perform functions such as echoing, character-erase, and line-kill. Assuming that the CANONPROC module in Figure *9-7* fulfills this need, the application program first opens a raw tty Stream. Then, the CANONPROC module is pushed above the driver to create a Stream of the form shown on the right of the figure. The driver is not aware that a module has been placed above it and therefore continues to send single character messages upstream. The module receives single character messages from the driver, processes the characters, and accumulates them into line strings. Each line is placed into a message and sent to the Stream head. The head now finds more than one character in the messages it receives from downstream.

Stream head implementation accommodates this change in format automatically and transfers the multiple-character data into user space. The Stream head also keeps track of messages partially transferred into user space (for example, when the current user `read()` buffer can only hold part of the current message). Downstream operation is not affected: the head sends, and the driver receives, multiple character messages.

Note that the Stream head provides the interface between the Stream and user process. Modules and drivers do not have to implement user interface functions other than open and close.

9.5. Basic User Level Functions

STREAMS System Calls

After a Stream has been opened, STREAMS-related system calls allow a user process to insert and delete (push and pop) modules. That process can then communicate with and control the operation of the Stream head, modules, and drivers, and can send and receive messages containing data and control information. This chapter presents an example of some of the basic functions available to STREAMS-based applications via the system calls. Additional functions are described at the end of this chapter and in the *Other Facilities* chapter.

The full set of STREAMS-related system calls is:

`open()`
Open a Stream

`close()`
Close a Stream

`read()`
Read data from a Stream

`write()`
Write data to a Stream

`ioctl()`
Control a Stream

`getmsg()`
Receive the message at Stream head

`putmsg()`
Send a message downstream

`poll()`
Notify the application program when selected events occur on a Stream

The following two-part example describes a Stream that controls the data communication characteristics of a connection between an asynchronous terminal and a tty port. It illustrates basic user level STREAMS features, then shows how messages can be used. The *Kernel Level Functions* chapter discusses the kernel Stream operations corresponding to the user operations described in this introduction.

An Asynchronous Protocol Stream Example

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

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

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

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

These options are set with the following modules:

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

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

ASCEBC

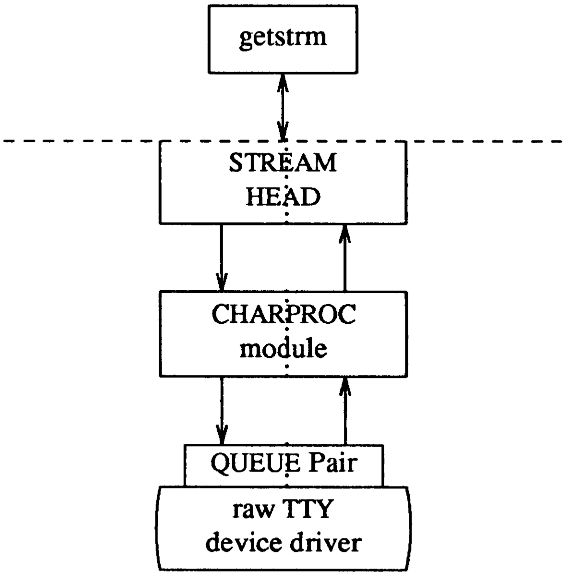
Translates EBCDIC code to ASCII upstream and ASCII to EBCDIC downstream.

Initializing the Stream

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

The initial idle Stream, shown in Figure *9-8*, contains only one pushable module, `CHARPROC`. The device driver is a limited function raw tty driver connected to a limited-function communication port. The driver and port transparently transmit and receive one unbuffered character at a time.

Figure 9-8 *Idle Stream Configuration for Example*



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

The next step is to analyze in more detail how the Stream sets up the communications options. Before doing so, let's examine how messages are handled in STREAMS.

Message Types

All STREAMS messages are assigned message types to indicate their intended use by modules and drivers and to determine their handling by the Stream head. A driver or module can assign most types to a message it generates, and a module can modify a message's type during processing. The Stream head will convert certain system calls to specified message types and send them downstream, and it will respond to other calls by copying the contents of certain message types that were sent upstream. Messages exist only in the kernel, so a user process can only send and receive buffers. The process is not explicitly aware of the message type, but it may be aware of message boundaries, depending on the system call used (see the distinction between `getmsg()` and `read()` in the next section).

Most message types are internal to STREAMS and can only be passed from one STREAMS module to another. A few message types, including `M_DATA`, `M_PROTO`, and `M_PCPROTO`, can also be passed between a Stream and user processes. `M_DATA` messages carry data within a Stream and between a Stream and a user process. `M_PROTO` or `M_PCPROTO` messages carry both data and control information. However, the distinction between control information and data is generally determined by the developer when implementing a particular Stream. Control information includes service interface information, carried between two Stream entities that present service interfaces, and condition or status information, which may be sent between any two Stream entities regardless of their interface. An `M_PCPROTO` message has the same general use as an `M_PROTO`, but the former moves faster through a Stream (see *Message Queue Priority* in the *Other Facilities* chapter).

Sending and Receiving Messages

`putmsg()` is a STREAMS-related system call that sends messages; it is similar to `write()`. `putmsg()` provides a data buffer which is converted into an `M_DATA` message, and can also provide a separate control buffer to be placed into an `M_PROTO` or `M_PCPROTO` block. `write()` provides byte-stream data to be converted into `M_DATA` messages.

`getmsg()` is a STREAMS-related system call that accepts messages; it is similar to `read()`. One difference between the two calls is that `read()` accepts only data (messages sent upstream to the Stream head as message type `M_DATA`), such as the characters entered from the terminal. `getmsg()` can simultaneously accept both data and control information (message sent upstream as types `M_PROTO` or `M_PCPROTO`). `getmsg()` also differs from `read()` in that it preserves message boundaries so that the same boundaries exist above and below the Stream head (that is, between a user process and a Stream). `read()` generally ignores message boundaries, processing data as a byte stream.

Certain STREAMS `ioctl()` commands, such as `I_STR`, also cause messages to be sent or received on the Stream. `I_STR` provides the general "ioctl" capability of the character I/O subsystem. A user process above the Stream head can issue `putmsg()`, `getmsg()`, the `I_STR` `ioctl()` command, and certain other STREAMS related system calls. Other STREAMS `ioctl's` perform functions that include changing the state of the Stream head, pushing and popping modules, or returning special information.

In addition to message types that explicitly transfer data to a process, some messages sent upstream result in information transfer. When these messages reach the Stream head, they are transformed into various forms and sent to the user process. The forms include signals, error codes, and call return values.

Using Messages in the Example

Returning to the asynchronous protocol example, the Stream was in its idle configuration (see Figure *9-8*). `getstrm()` had issued a `getmsg()` and was sleeping until the arrival of a message from the Stream head. Such a message would result from the driver detecting activity on the associated tty port.

An incoming call arrives at port one and causes a ring detect signal in the modem. The driver receives the ring signal, answers the call, and sends upstream an `M_PROTO` message containing information indicating an incoming call. `getstrm()` is notified of all incoming calls, although it can choose to refuse the call because of system limits. In this idle state, `getstrm()` will also accept `M_PROTO` messages indicating, for example, error conditions such as detection of line or modem problems on the idle line.

The `M_PROTO` message containing notification of the incoming call flows upstream from the driver into `CHARPROC`. `CHARPROC` inspects the message type, determines that message processing is not required, and passes the unmodified message upstream to the Stream head. The Stream head copies the message into the `getmsg()` buffers (one buffer for control information, the other for data) associated with `getstrm()` and wakes up the process. `getstrm()` sends its acceptance of the incoming call with a `putmsg()` system call which results in a downstream `M_PROTO` message to the driver.

Then, `getstrm()` sends a prompt to the operator with a `write()` and issues a `getmsg()` to receive the response. A `read()` could have been used to receive the response, but the `getmsg()` call allows concurrent monitoring for control (`M_PROTO` and `M_PCPROTO`) information. `getstrm()` will now sleep until the response characters, or information regarding possible error conditions detected by modules or driver, are sent upstream.

The first response, sent upstream in a `M_DATA` block, indicates that the code set is ASCII and that canonical processing is requested. `getstrm()` implements these options by pushing `CANONPROC` onto the Stream, above `CHARPROC`, to perform canonical processing on the input ASCII characters.

The response to the next prompt requests even parity checking. `getstrm()` sends an `ioctl()` `I_STR` command to `CHARPROC`, requesting the module to perform even parity checking on upstream characters. When the dialog indicate protocol option setting is complete, `getstrm()` creates an application process. At the end of the connection, `getstrm()` will pop `CANONPROC` and then send a `I_STR` to `CHARPROC` requesting the module to restore the no-parity idle state (`CHARPROC` remains on the Stream).

As a result of the above dialogs, the terminal at port one operates in the following configuration:

- ASCII, even parity

- Echo
- Canonical processing

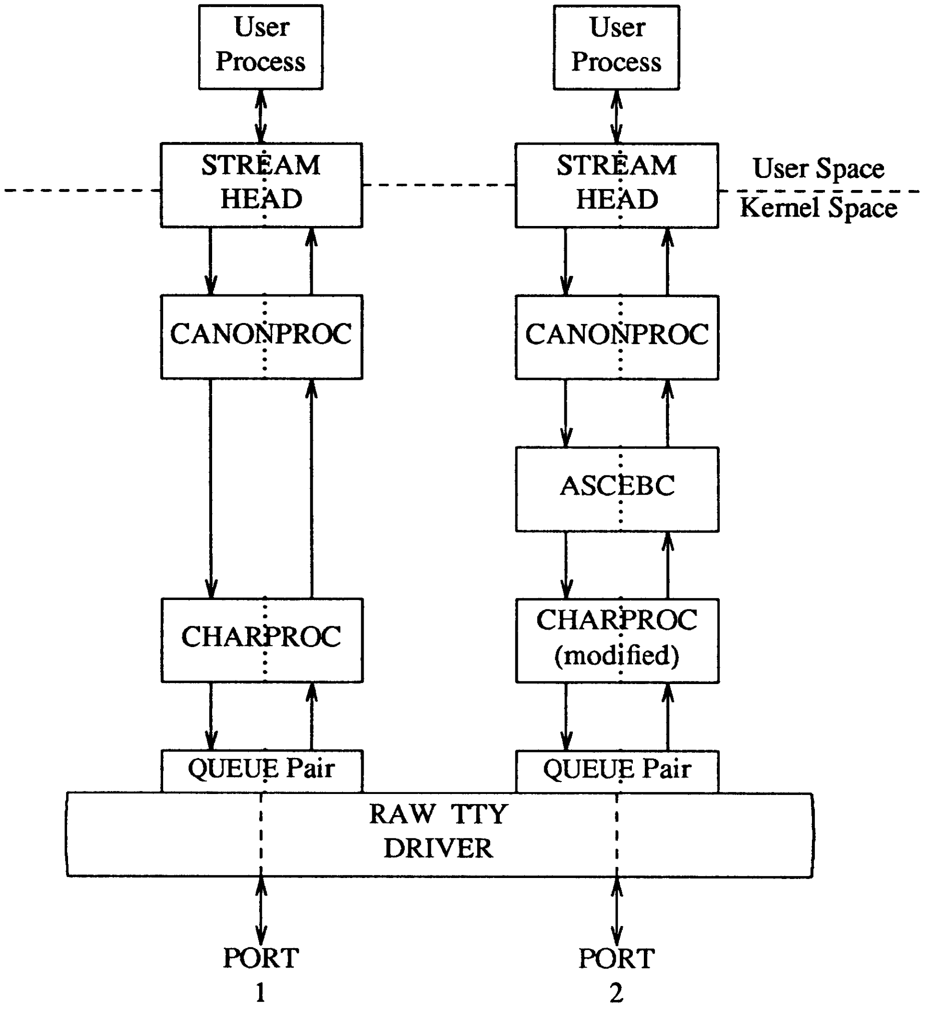
In similar fashion, an operator at a different type of terminal on port two requests a different set of options, resulting in the following configuration:

- EBCDIC
- No Echo
- Canonical processing

The resultant Streams for the two ports are shown in Figure *9-9*. For port one, on the left, the modules in the Stream are `CANONPROC` and `CHARPROC`.

For port two, on the right, the resultant modules are `CANONPROC`, `ASCEBC` and `CHARPROC`. `ASCEBC` has been pushed on this Stream to translate between the ASCII interface at the downstream side of `CANONPROC` and the EBCDIC interface of the upstream output side of `CHARPROC`. In addition, `getstrm()` has sent an `I_STR` to the `CHARPROC` module in this Stream requesting it to disable echo. The resultant modification to `CHARPROC's` functions is indicated by the word "modified" in the right Stream of Figure *9-9*.

Figure 9-9 *Asynchronous Terminal Streams*


User Process
User Process
STREAM HEAD
STREAM HEAD
User Space
Kernel Space
CANONPROC
CANONPROC
ASCEBC
CHARPROC
CHARPROC (modified)
QUEUE Pair
QUEUE Pair
RAW TTY DRIVER
PORT 1
PORT 2

Since `CHARPROC` is now performing no function for port two, it might have been popped from the Stream to be reinserted by `getstrm()` at the end of connection. However, the low overhead of STREAMS does not require its removal. The module remains on the Stream, passing messages unmodified between `ASCEBC` and the driver. At the end of the connection, `getstrm()` restores this Stream to its idle configuration of Figure *9-8* by popping the added modules and then sending an `I_STR` to `CHARPROC` to restore the echo default.

Note that the tty driver shown in Figure *9-9* handles minor devices. Each minor device has a distinct Stream connected from user space to the driver. This ability to handle multiple devices is a standard STREAMS feature, similar to the minor device mechanism in character I/O device drivers.

Other User Functions

The previous example illustrates basic STREAMS concepts. Alternate, more efficient, STREAMS calls or mechanisms could have been used in place of those described earlier. (Some of the alternatives are described in the *Other Facilities* chapter. For details, see following chapters and the *SunOS Reference Manual.*

For example, the initialization process that created a `getstrm()` for each tty port could have been implemented as a "supergetty" by use of the STREAMS-related `poll()` system call. As described in the *Other Facilities* chapter, `poll()` allows a single process to efficiently monitor and control multiple Streams. The "supergetty" process would handle all of the Stream and terminal protocol initialization and would create application processes only for established connections.

The `M_PROTO` notification sent to `getstrm()` could have been sent by the driver as an `M_SIG` message that causes a specified signal to be sent to the process. As discussed previously under *Message Types*, error and status information can also be sent upstream from a driver or module to user processes via different message types. These messages will be transformed by the Stream head into a signal or error code.

Finally, an `ioctl I_STR` command could have been used in place of a `putmsg M_PROTO` message to send information to a driver. The sending process must receive an explicit response from an `I_STR` by a specified time period or an error will be returned. A response message must be sent upstream by the destination module or driver to be translated into the user response by the Stream head.

9.6. Kernel Level Functions

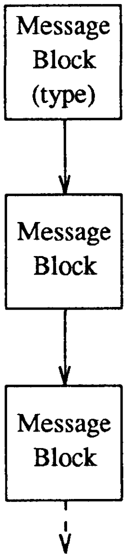
This chapter introduces the use of the STREAMS mechanism in the kernel and describes some of the tools provided by STREAMS to assist in the development of modules and drivers. In addition to the basic message passing mechanism and `QUEUE` Stream linkage described previously, the STREAMS mechanism consists of various facilities including buffer management, the STREAMS scheduler, processing and message priority, flow control, and multiplexing. Over 30 STREAMS utility routines and macros are available to manipulate and utilize these facilities.

The key elements of a STREAMS kernel implementation are the processing routines in the module and drivers, and the preparation of required data structures. The structures are described in the STREAMS section of *Writing Device Drivers.* The following sections provide further information on messages and on the processing routines that operate on them. The example of the previous chapter is continued, associating the user-level operations described there with kernel operations.

Messages

As shown in Figure *9-10*, a STREAMS message consists of one or more linked *message blocks*. That is, the first message block of a message may be attached to other message blocks that are part of the same message. Multiple blocks in a message can occur, for example, as the result of processing that adds header or trailer data to the data contained in the message, or because of message buffer size limitations which cause the data to span multiple blocks. When a message is composed of multiple message blocks, the message type of the first block determines the type of the entire message, regardless of the types of the attached message blocks.

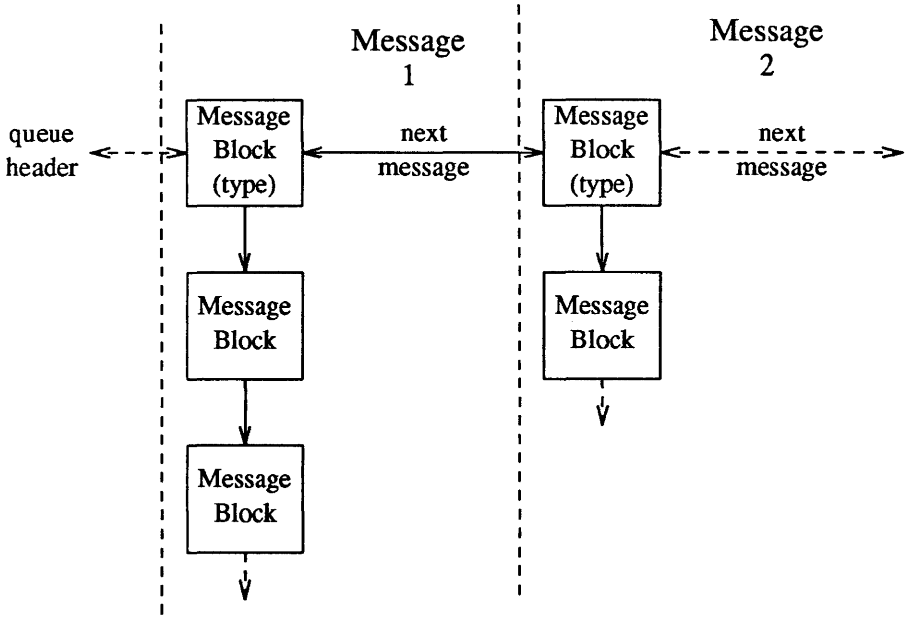
Figure 9-10 *A Message*



STREAMS allocates a message as a single block containing a buffer of a certain size (see the next section). If the data for a message exceed the size of the buffer containing the data, the procedure can allocate a new block containing a larger buffer, copy the current data to it, insert the new data and de-allocate the old block. Alternately, the procedure can allocate an additional (smaller) block, place the new data in the new message block and link it after or before the initial message block. Both alternatives yield one new message.

Messages can exist standalone, as shown in Figure *9-10* when the message is being processed by a procedure. Alternately, a message can await processing on a linked list of messages, called a message queue, in a `QUEUE`. In Figure *9-11*, Message 1 is linked to Message 2.

Figure 9-11 *Messages on a Message Queue*



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

STREAMS utility routines enable developers to manipulate messages and message queues.

Message Allocation

STREAMS maintains its own storage pool for messages. A procedure can request the allocation of a message of a specified size at one of three message pool priorities. The `allocb()` utility will return a message containing a single block with a buffer of at least the size requested, providing there is a buffer available at the priority requested. When requesting priority for messages, developers must weigh their process' need for resources against the needs of other processes on the same machine.

Message pool priority generally has no effect on allocation until the pool falls below internal STREAMS thresholds. When this occurs, `allocb()` may refuse a lower priority request for a message of size "x" while granting a higher priority request for the same size message. As examples of priority usage, storage for an urgent control message, such as an `M_HANGUP` or `M_PCPROTO` could be requested at high priority. An `M_DATA` buffer for holding input might be requested at medium priority, and an output buffer (presuming the output data can wait in user space) at lowest priority.

Put and Service Procedures

The procedures in the QUEUE are the software routines that process messages as they transit the QUEUE. The processing is generally performed according to the message type and can result in a modified message, new message(s) or no message. A resultant message is generally sent in the same direction in which it was received by the QUEUE, but may be sent in either direction. A QUEUE will always contain a put procedure and may also contain an associated service procedure.

Put Procedures

A put procedure is the QUEUE routine that receives messages from the preceding QUEUE in the Stream. Messages are passed between QUEUEs by a procedure in one QUEUE calling the put procedure contained in the following QUEUE. A call to the put procedure in the appropriate direction is generally the only way to pass messages between modules (unless otherwise indicated, "modules" infers "module, driver and Stream head"). QUEUEs in pushable (see *Building a Stream*) modules contain a put procedure. In general, there is a separate put procedure for the read and write QUEUEs in a module because of the "full duplex" operation of most Streams.

A put procedure is associated with immediate (as opposed to deferred, see below) processing on a message. Each module accesses the adjacent put procedure as a subroutine. For example, consider that *modA*, *modB*, and *modC* are three consecutive modules in a Stream, with *modC* connected to the Stream head. If *modA* receives a message to be sent upstream, *modA* processes that message and calls *modB*'s put procedure, which processes it and calls *modC*'s put procedure, which processes it and calls the Stream head's put procedure. Thus, the message will be passed along the Stream in one continuous processing sequence. On one hand, this sequence has the benefit of completing the entire processing in a short time with low overhead (subroutine calls). On the other hand, if this sequence is lengthy and the processing is implemented on a multi-user system, then this manner of processing may be good for this Stream but may be detrimental for others since they may have to wait "too long" to get their turn at bat.

In addition, there are situations where the put procedure cannot immediately process the message but must hold it until processing is allowed. The most typical examples of this are a driver which must wait until the current output completes before sending the next message and the Stream head, which may have to wait until a process initiates a `read(2)` on the Stream.

Service Procedures

STREAMS allows a service procedure to be contained in each QUEUE, in addition to the put procedure, to address the above cases and for additional purposes. A service procedure is not required in a QUEUE and is associated with deferred processing. If a QUEUE has both a put and service procedure, message processing will generally be divided between the procedures. The put procedure is always called first, from a preceding QUEUE. After the put procedure completes its part of the message processing, it arranges for the service procedure to be called by passing the message to the `putq()` routine. `putq()` does two things: it places the message on the message queue of the QUEUE (see Figure *9-11*) and links the QUEUE to the end of the STREAMS scheduling queue. When `putq()` returns to the put procedure, the procedure typically exits. Some time later, the service procedure will be automatically called by the STREAMS scheduler.

The STREAMS scheduler is separate and distinct from the SunOS system process scheduler. It is concerned only with QUEUEs linked on the STREAMS scheduling queue. The scheduler calls the service procedure of the scheduled QUEUE in a FIFO manner, one at a time.

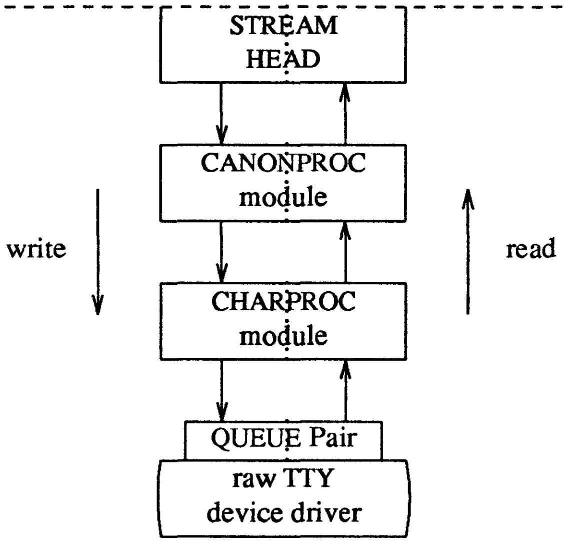
Having both a put and service procedure in a QUEUE enables STREAMS to provide the rapid response and the queuing required in multi-user systems. The put procedure allows rapid response to certain data and events, such as software echoing of input characters. Put procedures effectively have higher priority than any scheduled service procedures. When called from the preceding STREAMS component, a put procedure executes before the scheduled service procedures of any QUEUE are executed.

The service procedure implies message queuing. Queuing results in deferred processing of the service procedure, following all other QUEUEs currently on the scheduling queue. For example, terminal output and input erase and kill processing would typically be performed in a service procedure because this type of processing does not have to be as timely as echoing. Use of a service procedure also allows processing time to be more evenly spread among multiple Streams. As with the put procedure there will generally be a separate service procedure for each QUEUE in a module. The flow control mechanism (see the *Other Facilities* chapter) uses the service procedures.

Kernel Processing

The following continues the example of the previous chapter, describing STREAMS kernel operations and associating them, where relevant, with the user-level system calls already discussed. As a result of initializing operations and pushing a module, the Stream for port one has the following configuration:

Figure 9-12 *Operational Stream for Example*



As shown in Figure *9-12* the upstream QUEUE is also referred to as the read QUEUE, reflecting the message flow in response to a `read()` system call. Correspondingly, downstream is referred to as the write QUEUE. Read side processing is discussed first.

Read Side Processing

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

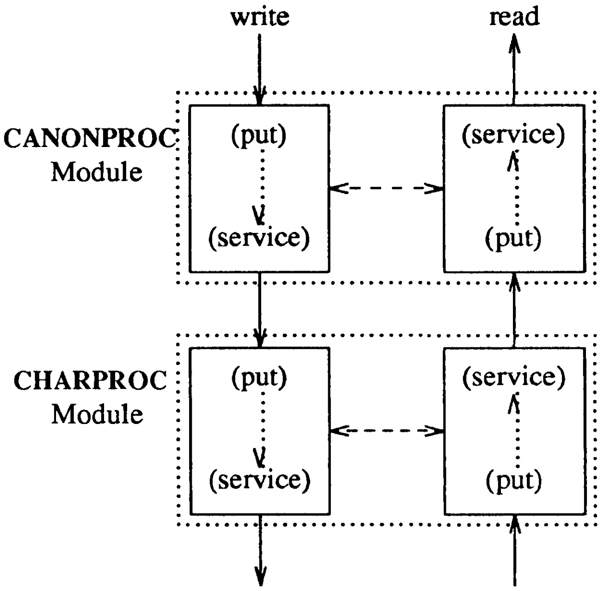
Driver Processing

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

CHARPROC

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

Figure 9-13 *Module Put and Service Procedures*



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

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

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

CANONPROC

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

Write Side Processing

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

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

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

Analysis

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

9.7. Other Facilities

The previous chapters described the basic concepts of constructing a Stream and utilizing the STREAMS mechanism. Additional STREAMS features are provided to handle characteristic problems of protocol implementation, such as flow control, and to assist in development.

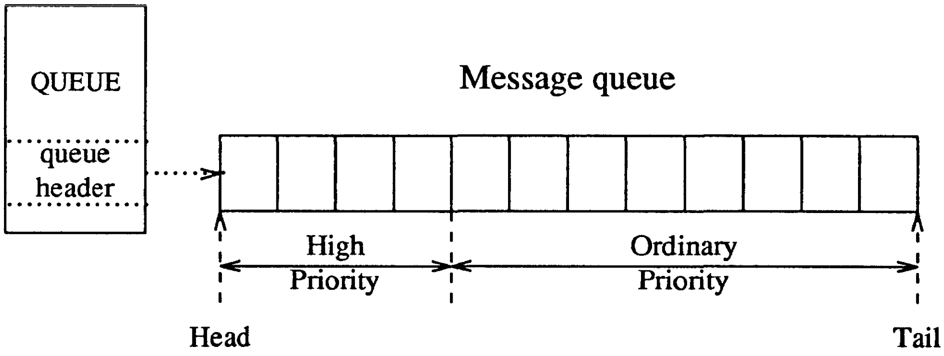
There are also kernel and user-level facilities that support the implementation of advanced functions, such as multiplexors, and allow asynchronous operation of a user process and STREAMS input and output.

Message Queue Priority

As mentioned in the previous chapter, the STREAMS scheduler operates strictly FIFO so that each `QUEUE`'s service procedure receives control in the order it was scheduled. When a service procedure receives control, it may encounter multiple messages on its message queue. This buildup can occur if there is a long interval between the time a message is queued by a put procedure and the time that the STREAMS scheduler calls the associated service procedure. In this interval, there can be multiple calls to the put procedure causing multiple messages. The service procedure always processes all messages on its message queue unless prevented by flow control (see next section). Each message must pass through all the modules connecting its origin and destination in the Stream.

If service procedures were used in all `QUEUE`s and there was no message priority, then the most recently scheduled message would be processed after all the other scheduled messages on all Streams had been processed. In certain cases, message types containing urgent information (such as a break or alarm conditions) must pass through the Stream quickly. To accommodate these cases, STREAMS provides two classes of message queuing priority, ordinary and high. STREAMS prevents high-priority messages from being blocked by flow control and causes a service procedure to process them ahead of all ordinary priority messages on the procedure's queue. This results in the high-priority message transiting each module with minimal delay.

Figure 9-14 *Streams Message Priority*



The priority mechanism operates as shown in Figure *9-14*. Message queues are generally not present in a `QUEUE` unless that `QUEUE` contains a service procedure. When a message is passed to `putq()` to schedule the message for service procedure processing, `putq()` places the message on the message queue in priority order. High priority messages are placed ahead of all ordinary priority messages, but behind any other high priority messages on the queue. STREAMS utilities deliver the messages to the processing service procedure FIFO within each priority class. The service procedure is unaware of the message priority and simply receives the next message.

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

Flow Control

Even on a well-designed system, general system delays, malfunctions, and excessive message accumulation on one or more Streams can cause the message buffer pools to become depleted. Additionally, processing bursts can arise when a service procedure in one module has a long message queue and processes all its messages in one pass. STREAMS provides two independent mechanisms to guard its message buffer pools from being depleted and to minimize long processing bursts at any one module.

Flow control is only applied to normal priority messages (see previous section) and not to high priority messages.

The first flow control mechanism is global and automatic. When the Stream head requests a message buffer in response to a `putmsg()` or `write()` system call, it uses the lowest level of priority. Since buffer availability is based on priority and buffer pool levels, the Stream head will be among the first modules refused a buffer when the pool becomes depleted. In response, the Stream head will block user output until the STREAMS buffer pool recovers. As a result, output has a lower priority than input.

The second flow control mechanism is local to each Stream and advisory (voluntary), and limits the number of characters that can be queued for processing at any `QUEUE` in a Stream. This mechanism limits the buffers and related processing at any one QUEUE and in any one Stream, but does not consider buffer pool levels or buffer usage in other Streams.

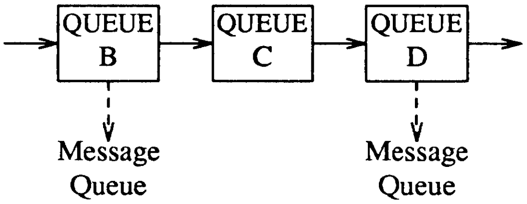
The advisory mechanism operates between the two nearest QUEUEs in a Stream containing service procedures (see diagram on next page). Messages are generally held on a message queue only if a service procedure is present in the associated QUEUE.

Messages accumulate at a QUEUE when its service procedure processing does not keep pace with the message arrival rate, or when the procedure is blocked from placing its messages on the following Stream component by the flow control mechanism. Pushable modules contain independent upstream and downstream limits, which are set when a developer specifies high-water and low-water control values for the QUEUE. The Stream head contains a preset upstream limit (which can be modified by a special message sent from downstream) and a driver may contain a downstream limit.

Flow control operates as follows:

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

Figure 9-15 *Flow Control*



- The service procedure in B uses a STREAMS utility routine to see if a QUEUE ahead is marked full. If messages cannot be sent, the scheduler blocks the service procedure in B from further execution. B remains blocked until the low-water mark of the full QUEUE, D, is reached.
- While B is blocked, any non-priority messages that arrive at B will accumulate on its message queue (recall that priority messages are not blocked). In turn, B can reach a full state and the full condition will propagate back to the last module in the Stream.

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

Note that to utilize flow control, a developer need only call the utility that tests if a full condition exists ahead, plus perform some housekeeping if it does. Everything else is automatically handled by STREAMS.

Multiplexing

STREAMS multiplexing supports the development of internetworking protocols such as IP and ISO CLNS, and the processing of interleaved data streams such as in SNA, X.25, and terminal window facilities.

STREAMS multiplexors (also called pseudo-device drivers) are created in the kernel by interconnecting multiple Streams. Conceptually, there are two kinds of multiplexors that developers can build with STREAMS: upper and lower multiplexors. Lower multiplexors have multiple lower Streams between device drivers and the multiplexor, and upper multiplexors have multiple upper Streams between user processes the multiplexor.

Figure 9-16 *Internet Multiplexing Stream*

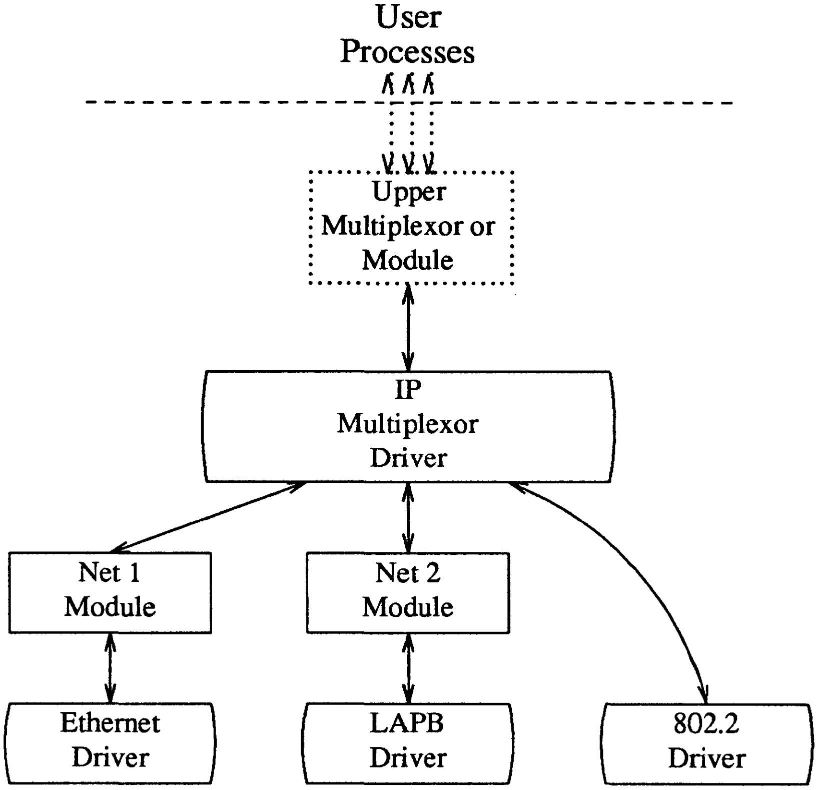


Figure *9-16* shows an example of a lower multiplexor. This configuration would typically occur where internetworking functions were included in the system. This Stream contains two types of drivers: the Ethernet, LAPB, and IEEE 802.2

are hardware device drivers that terminate links to other nodes; the IP (Internet Protocol) is a multiplexor.

The IP multiplexor switches messages among the various nodes (lower Streams) or sends them upstream to user processes in the system. In this example, the multiplexor expects to see an 802.2 interface downstream; for the Ethernet and LAPB drivers, the Net 1 and Net 2 modules provide service interfaces to the two the non-802.2 drivers and the IP multiplexor.

Figure *9-16* depicts the IP multiplexor as part of a larger Stream. The Stream, as shown in the dotted rectangle, would generally have an upper TCP multiplexor and additional modules. Multiplexors could also be cascaded below the IP driver if the device drivers were replaced by multiplexor drivers.

Figure 9-17 *X.25 Multiplexing Stream*

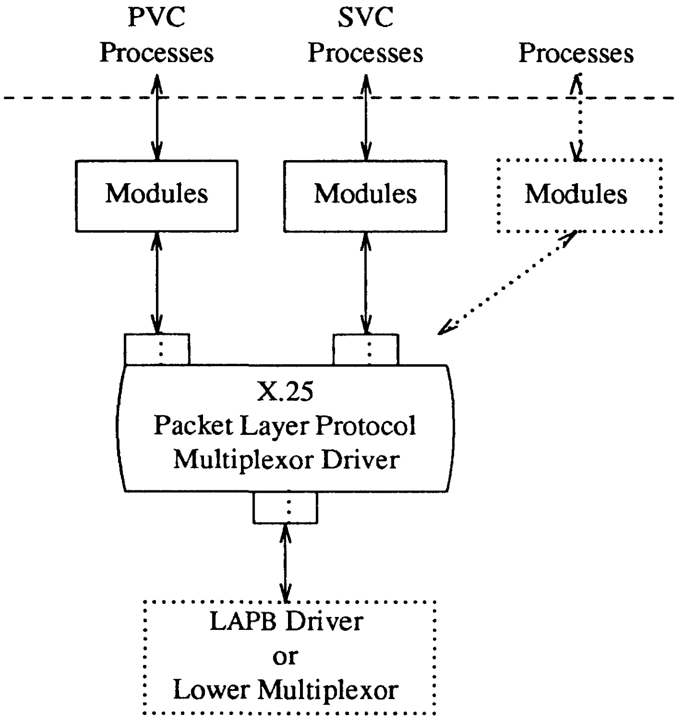


Figure *9-17* shows an upper multiplexor. In this configuration, the driver routes messages between the lower Stream and one of the upper Streams. This Stream performs X.25 multiplexing to multiple independent SVC (Switched Virtual Circuit) and PVC (Permanent Virtual Circuit) user processes. Upper multiplexors are a specific application of standard STREAMS facilities that support multiple minor devices in a device driver. This figure also shows that more complex configurations can be built by having one or more multiplexed LAPB drivers below and multiple modules above.

Developers can choose either upper or lower multiplexing, or both, when designing their applications. For example, a window multiplexor would have a similar configuration to the X.25 configuration of Figure *9-16*, with a window driver replacing Packet Layer, a tty driver replacing LAPB, and the child processes of the terminal process replacing the user processes. Although the X.25 and window multiplexing Streams have similar configurations, their multiplexor drivers would differ significantly. The IP multiplexor of Figure *9-15* has a different configuration than the X.25 multiplexor and the driver would implement its own set of processing and routing requirements.

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

The connections are created from user space through specific STREAMS `ioctl()` system calls. In a lower multiplexor, multiple Streams are connected below an application-specific, developer-implemented multiplexing driver. The multiplexing facility will only connect Streams to a driver. The `ioctl()` call configures a multiplexor by connecting one Stream at a time below the opened multiplexor driver. As each Stream is connected to the driver, the connection setup procedure identifies the Stream to the driver. The driver will generally store this setup information in a private data structure for later use.

Subsequently, when messages flow into the driver on the various connected Streams, the identity of the associated Stream is passed to the driver as part of the standard procedure call. The driver then has available the Stream identification, the previously stored setup information for this Stream, and any internal routing information contained in the message. These data are used, according to the application implemented, to process the incoming message and route the output to the appropriate outgoing Stream.

Additionally, new Streams can be dynamically connected to a operating multiplexor without interfering with ongoing traffic, and existing Streams can be disconnected with similar ease.

Monitoring

STREAMS allows user processes to monitor and control Streams so that system resources (such as CPU cycles and process slots) can be used effectively. Monitoring is especially useful to user-level multiplexors, in which a user process can create multiple Streams and switch messages among them (similar to STREAMS kernel-level multiplexing, described previously).

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

Synchronous monitoring is provided by use of `poll()` alone; in this case, the user process cannot continue processing until after the system call completes. When the calls are used together, they allow asynchronous, or concurrent, operation of the process and STREAMS input/output. This allows the user process to monitor the Stream while carrying on other activities.

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

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

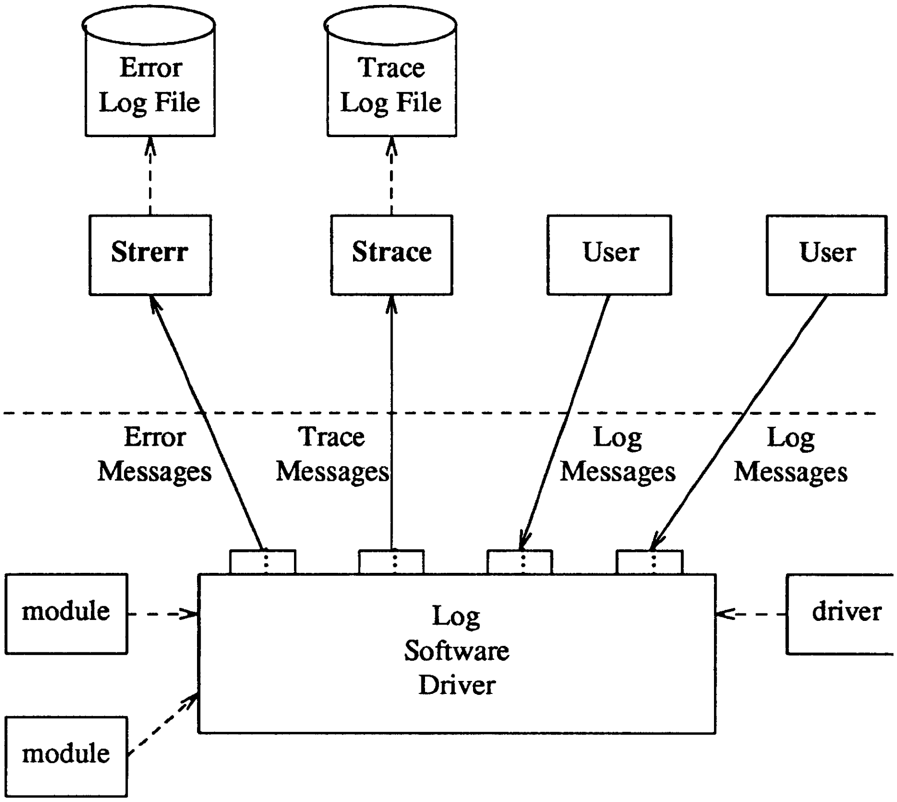
If any selected event occurs on any of the selected Streams, STREAMS will cause the `SIGPOLL` catching function to be executed in all associated requesting processes. However, the process(es) will not know which event occurred, nor on what Stream the event occurred. A process that issues the `I_SETSIG`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

Error and Trace Logging

STREAMS includes error and trace loggers useful for debugging and administering modules and drivers.

Any module or driver in any Stream can call the STREAMS logging function `strlog()`, described in `log`(4). When called, `strlog()` will send formatted text to the error logger `strerr`(8V), the trace logger `strace`(8V), or both. The call parameters for `strlog()` include the module/driver identification, a severity level, and the formatted text describing the condition causing the call. The call also identifies the process (`strerr()` and/or `strace()`) to receive the resultant output message.

Figure 9-18 *Error and Trace Logging*


Error Log File
Trace Log File
Strerr
Strace
User
User
Error Messages
Trace Messages
Log Messages
Log Messages
module
module
Log Software Driver
driver

`strerr()` is intended to operate as a daemon process initiated at system startup. A call to `strlog()` requesting an error to be logged causes an `M_PROTO` message to be sent to `strerr()`, which formats the contents and places them in a daily file. The utility `strclean(8V)` is provided to periodically purge aged, unreferenced daily log files.

A call to `strlog()` requesting trace information to be logged causes a similar `M_PROTO` message to be sent to `strace(8V)`, which places it in a user designated file. `strace()` is intended to be initiated by a user. The user can designate the modules/drivers and severity level of the messages to be accepted for logging by `strace()`.

A user process can submit its own `M_PROTO` messages to the log driver for inclusion in the logger of its choice through `putmsg(2)`. The messages must be in the same format required by the logging processes and will be switched to the logger(s) requested in the message.

The output to the log files is formatted, ASCII text. The files can be processed by standard system commands such as `grep(1)` or `ed(1)`, or by developer-provided routines.

9.8. Driver Design Comparisons

This chapter compares operational features of character I/O device drivers with STREAMS drivers and modules. It is intended for experienced developers of UNIX system character device drivers. Details are provided in the STREAMS section of *Writing Device Drivers*.

Environment

No user environment is generally available to STREAMS module procedures and drivers. The exception is the module and driver open and close routines, both of which have access to the `user` structure of the calling process and can sleep. Otherwise, a STREAMS driver, module put procedure, and module service procedure has no user context and can neither sleep nor access any `user` structure.

Multiple Streams can use a copy of the same module (that is, the same `fmodsw`), each containing the same processing procedures. This means that module code is reentrant, so care must be exercised when using global data in a module. Put and service procedures are always passed the address of the `QUEUE` (for example, in Figure *9-6* Au calls Bu's put procedure with Bu as a parameter). The processing procedure establishes its environment solely from the `QUEUE` contents, typically the private data (for example, state information).

Drivers

At the interface to hardware devices, character I/O drivers have interrupt entry points; at the system interface, those same drivers generally have direct entry points (routines) to process `probe()`, `open()`, `close()`, `read()`, `write()` and `ioctl()` system calls.

STREAMS device drivers have similar interrupt entry points at the hardware device interface and have direct entry points only for `open()` and `close()` system calls. These entry points are accessed via STREAMS, and the call formats differ from character device drivers. The put procedure is a driver's third entry point, but it is a message (not system) interface. The Stream head translates `write()` and `ioctl()` calls into messages and sends them downstream to be processed by the driver's write `QUEUE` put procedure. `read()` is seen directly only by the Stream head, which contains the functions required to process system calls. A driver does not know about system interfaces other than `open()` and `close()`, but it can detect absence of a `read()` indirectly if flow control propagates from the Stream head to the driver and affects the driver's ability to send messages upstream.

For input processing, when the driver is ready to send data or other information to a user process, it does not wake up the process. It prepares a message and sends it to the read `QUEUE` of the appropriate (minor device) Stream. The driver's open routine generally stores the `QUEUE` address corresponding to this Stream.

For output processing, the driver receives messages in place of a `write()` call. If the message can not be sent immediately to the hardware, it may be stored on the driver's write message queue. Subsequent output interrupts can remove messages from this queue.

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

Modules

As described above, modules have user context available only during the execution of their open and close routines. Otherwise, the `QUEUE`s forming the module are not associated with the user process at the end of the Stream, nor with any other process. Because of this, `QUEUE` procedures must not sleep when they cannot proceed; instead, they must explicitly return control to the system. The system saves no state information for the `QUEUE`. The `QUEUE` must store this information internally if it is to proceed from the same point on a later entry.

When a module or driver that requires private working storage (for example, for state information) is pushed, the open routine must obtain the storage from external sources. STREAMS copies the module template from `fmodsw` for the `I_PUSH`, so only fixed data can be contained in the module template. STREAMS has no automatic mechanism to allocate working storage to a module when it is opened. The sources for the storage typically include a module-specific kernel array, installed when the system is configured, or the STREAMS buffer pool. When using an array as a module storage pool, the maximum number of copies of the module that can exist at any one time must be determined. For drivers, this is typically determined from the physical devices connected, such as the number of ports on a multiplexor. However, certain types of modules may not be associated with a particular external physical limit. For example, the CANONICAL module shown in Figure *9-5* could be used on different types of Streams.

9.9. Glossary

Downstream	The direction from Stream head to driver.
Driver	The end of the Stream closest to an external interface. The principal functions of the driver are handling any associated device, and transforming data and information between the external interface and Stream. It can also be a pseudo-driver, not directly associated with a device, which performs functions internal to a Stream, such as a multiplexor or log driver.
Message	One or more linked blocks of data or information, with associated STREAMS control structures containing a message type. Messages are the only means of transferring data and communicating within a Stream.
Message Queue	A linked list of messages connected to a `QUEUE`.
Message Type	A defined set of values identifying the contents of a message.
Module	Software that performs functions on messages as they flow between Stream head and driver. A module is the STREAMS counterpart to the commands in a Shell pipeline except that a module contains a pair of functions which allow independent bidirectional (downstream and upstream) data flow and processing.
Multiplexor	A mechanism for connecting multiple Streams to a multiplexing driver. The mechanism supports the processing of interleaved data Streams and the processing of internetworking protocols. The multiplexing driver routes messages among the connected Streams. The other end of a Stream connected to a multiplexing driver is typically connected to a device driver.

pushable module	A module between the Stream head and driver. A driver is a non-pushable module and a Stream head includes a non-pushable module.
QUEUE	The set of structures that forms a module. A module is composed of two `QUEUE`s, a read (upstream) `QUEUE` and a write (downstream) `QUEUE`.
Read Queue	The message queue in a module or driver containing messages moving upstream. Associated with input from a driver.
Stream	The kernel aggregate created by connecting STREAMS components, resulting from an application of the STREAMS mechanism. The primary components are a Stream head, a driver and zero or more pushable modules between the Stream head and driver. A Stream forms a full duplex processing and data transfer path in the kernel, between a user process and a driver. A Stream is analogous to a Shell pipeline except that data flow and processing are bidirectional.
Stream Head	The end of the Stream closest to the user process. The Stream head provides the interface between the Stream and the user process. The principal functions of the Stream head are processing STREAMS-related system calls, and bidirectional transfer of data and information between a user process and messages in STREAMS' kernel space.
STREAMS	A kernel mechanism that supports development of network services and data communication drivers. It defines interface standards for character input/output within the kernel, and between the kernel and user level. The STREAMS mechanism comprises integral functions, utility routines, kernel facilities and a set of structures.
Upstream	The direction from driver to Stream head.
Write Queue	The message queue in a module or driver containing messages moving downstream. Associated with output from a user process.

10

STREAMS Applications Programming

STREAMS Applications Programming **201**

10.1. Introduction 201
- Streams Overview 201
- Development Facilities 203

10.2. Basic Operations 204
- A Simple Stream 204
- Inserting Modules 206
- Module and Driver Control 207

10.3. Advanced Operations 210
- Advanced Input/Output Facilities 210
- Input/Output Polling 210
- Asynchronous Input/Output 213
- Clone Open 214

10.4. Multiplexed Streams 214
- Multiplexor Configurations 214
- Building a Multiplexor 216
- Dismantling a Multiplexor 221
- Routing Data Through a Multiplexor 222

10.5. Message Handling 223
- Service Interface Messages 223
  - Service Interfaces 223
- The Message Interface 224
- Datagram Service Interface Example 226

Accessing the Datagram Provider 228
Closing the Service 231
Sending a Datagram 231
Receiving a Datagram 232

10

STREAMS Applications Programming

10.1. Introduction

This chapter provides detailed information about the STREAMS mechanism and system call interface. It includes the following topics.

- The *Streams Overview*, below, reintroduces the STREAMS mechanism.
- *Basic Operations* describes the basic operations available for constructing, using, and dismantling streams. These operations are performed using `open`(2), `close`(2), `read`(2), `write`(2), and `ioctl`(2).
- *Advanced Operations* presents advanced facilities provided by STREAMS, including: `poll`(2), a user level I/O polling facility; asynchronous I/O processing support; and a method to sample drivers for available resources.
- *Multiplexed Streams* describes the construction of sophisticated, multiplexed stream configurations.
- *Message Handling* describes how users can process STREAMS messages using `putmsg(2)` and `getmsg(2)` in the context of a service interface example.

The following *STREAMS Module and Driver Programming* chapter is the companion to this chapter—it provides an analogous discussion of system-level STREAMS. Both chapters assume a working knowledge of UNIX† system programming, data communication facilities, and the material covered in the previous *Introduction to STREAMS* chapter.

Streams Overview

This section reviews the STREAMS mechanism, a general, flexible facility and a set of tools for development of SunOS and UNIX system communication services. It supports the implementation of services ranging from complete networking protocol suites to individual device drivers. The STREAMS mechanism defines standard interfaces for character I/O within the kernel, and between the kernel and the rest of the system. The associated mechanism is simple and open-ended. It consists of a set of system calls, kernel resources, and kernel routines.

The standard mechanism enables modular, portable development and easy integration of higher performance network services and their components. STREAMS provide a framework; they do not impose any specific network architecture. The STREAMS user interface is upward compatible with the character I/O user interface, and both user interfaces are available.

---

† UNIX is a registered trademark of AT&T.

A stream is a full-duplex processing and data transfer path between a STREAMS driver in kernel space and a process in user space (see the figure below). In the kernel, a stream is constructed by linking a stream head, a driver, and zero or more modules between the stream head and driver. The stream head is the end of the stream closest to the user process. Throughout this guide, the word "STREAMS" refers to the mechanism, and the word *stream* refers to the data path between a user and a driver.[7]

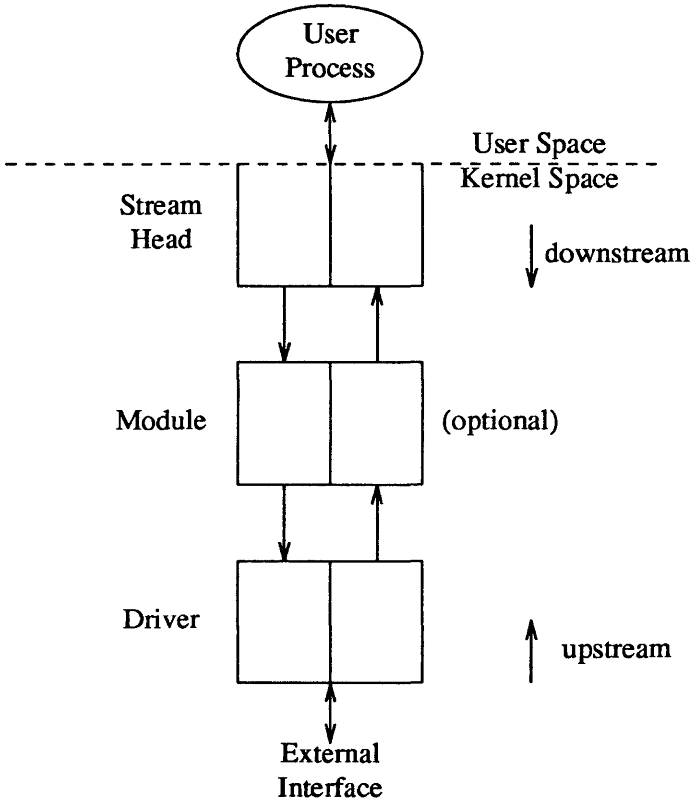
A STREAMS driver may be a device driver that provides the services of an external I/O device, or a software driver, commonly referred to as a pseudo-device driver, that performs functions internal to a stream. The stream head provides the interface between the stream and user processes. Its principal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

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

---

[7] The word "stream" is also used by 4.x BSD to refer to a nonseekable data source such as a pipe or socket. A STREAMS *stream* need not be restricted in this way.

Figure 10-1 *Basic Stream*



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

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

Development Facilities

General and STREAMS-specific system calls provide the user level facilities required to implement application programs. This system call interface is upwardly compatible with the character I/O facilities. The `open(2)` system call will recognize a STREAMS file and create a stream to the specified driver. A user process can receive and send data on STREAMS files using `read(2)` and `write(2)` in the same manner as with character files. The `ioctl(2)` system call enables users to perform functions specific to a particular device and a set of generic STREAMS `ioctl()` commands, described by `streamio`(4), support a variety of functions for accessing and controlling streams. A `close(2)` dismantles a stream.

In addition to the generic `ioctl()` commands, there are STREAMS-specific system calls to support unique STREAMS facilities. The `poll(2)` system call enables a user to poll multiple streams for various events. The `putmsg(2)` and `getmsg(2)` system calls enable users to send and receive STREAMS messages, and are suitable for interacting with STREAMS modules and drivers through a service interface.

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

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

10.2. Basic Operations

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

A Simple Stream

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

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

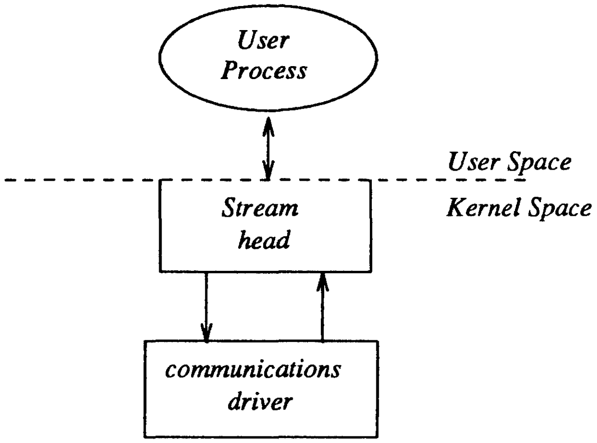
The `close()` system call will close a device and dismantle the associated stream.

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

```
#include <fcntl.h>

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

    if ((fd = open("/dev/comm01", O_RDWR)) < 0) {
        perror("open failed");
        exit(1);
    }
    while ((count = read(fd, buf, 1024)) > 0) {
        if (write(fd, buf, count) != count) {
            perror("write failed");
            break;
        }
    }
    exit(0);
}
```

In the example, `/dev/comm01` identifies a minor device of the communications device driver. When this file is opened, the system recognizes the device as a STREAMS device and connects a stream to the driver. The figure below shows the state of the stream following the call to `open()`.

Figure 10-2 *Stream to Communications Driver*



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

The `read()` call returns the available data, which may contain fewer than 1024 bytes. If no data are currently available at the stream head, the `read()` call blocks until data arrive.

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

Inserting Modules

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

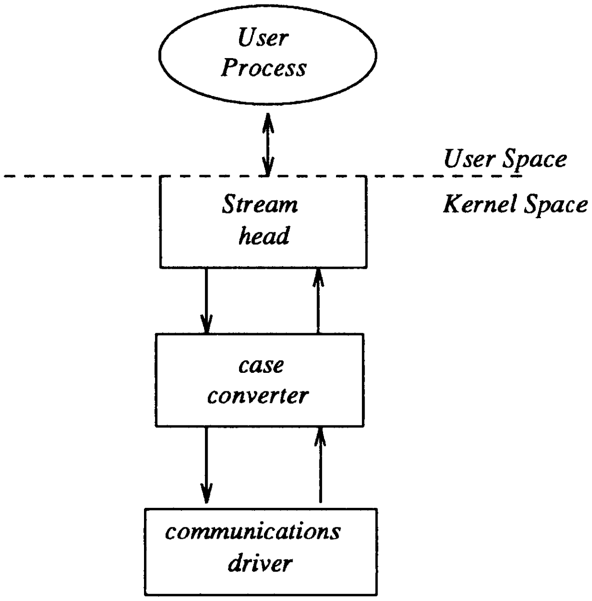
```
#include <string.h>
#include <fcntl.h>
#include <stropts.h>
/*
 * These defines would typically be
 * found in a header file for the module
 */
#define OUTPUT_STRING   1
#define INPUT_STRING    2

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

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

```
    if ((fd = open("/dev/comm01", O_RDWR)) < 0) {
        perror("open failed");
        exit(1);
    }
    if (ioctl(fd, I_PUSH, "case_converter") < 0) {
        perror("ioctl I_PUSH failed");
        exit(2);
    }
```

The I_PUSH `ioctl()` call directs the stream head to insert the case converter module between the driver and the stream head, creating the stream shown in the figure below. As with any driver, this module resides in the kernel and must have been configured into the system before it was booted. I_PUSH is one of several generic STREAMS `ioctl()` commands that enable a user to access and control individual streams (see the `streamio`(4) man page).

Figure 10-3 *Case Converter Module*



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

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

Module and Driver Control

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

```
/* Set input conversion string */
strioctl.ic_cmd = INPUT_STRING;        /* Command type */
strioctl.ic_timout = 0;                /* Default = 15 sec */
strioctl.ic_dp = "ABCDEFGHIJ";
strioctl.ic_len = strlen(strioctl.ic_dp);

if (ioctl(fd, I_STR, &strioctl) < 0) {
    perror("ioctl I_STR failed");
    exit(3);
}
/* Set output conversion string */
strioctl.ic_cmd = OUTPUT_STRING;       /* Command type */
strioctl.ic_dp = "abcdefghij";
strioctl.ic_len = strlen(strioctl.ic_dp);

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

`ioctl()` requests are issued to STREAMS drivers and modules indirectly, using the I_STR `ioctl()` call (see the `streamio`(4) man page). The argument to I_STR must be a pointer to a `strioctl` structure, which specifies the request to be made to a module or driver. This structure is defined in `<stropts.h>` and has the following format:

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

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

I_STR is intercepted by the stream head, which packages it into a message, using information contained in the `strioctl` structure, and sends the message downstream. The request will be processed by the module or driver closest to the stream head that understands the command specified by *ic_cmd*. The `ioctl()` call will block up to *ic_timout* seconds, waiting for the target module or driver to respond with either a positive or negative acknowledgement message. If an acknowledgement is not received in *ic_timout* seconds, the `ioctl()` call will fail.

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

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

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

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

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

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

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

As with the previous example, the `exit()` system call will dismantle the stream before terminating the process. The case converter module will be removed from the stream automatically when it is closed. Alternatively, modules may be removed from a stream using the I_POP `ioctl()` call described in `streamio`(4). This call removes the topmost module on the stream, and enables a user process to alter the configuration of a stream dynamically, by pushing and popping modules as needed.

A few of the important `ioctl()` requests supported by STREAMS have been discussed. Several other requests are available to support operations such as determining if a given module exists on the stream, or flushing the data on a stream. These requests are described fully in the `streamio`(4) man page).

10.3. Advanced Operations

This section introduces advanced features provided by STREAMS, such as an I/O polling facility, asynchronous I/O processing support, and a method to sample drivers for available resources.

Advanced Input/Output Facilities

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

Input/Output Polling

The `poll(2)` system call provides users with a mechanism for monitoring input and output on a set of file descriptors that reference open streams. It identifies those streams over which a user can send or receive data. For each stream of interest users can specify one or more events about which they should be notified. These events include the following:

`POLLIN`
: Input data are available on the stream associated with the given file descriptor.

`POLLPRI`
: A priority message is available on the stream associated with the given file descriptor. Priority messages are described in the section of Chapter 4 entitled "Accessing the Datagram Provider."

`POLLOUT`
: The stream associated with the given file is writable. That is, the stream has relieved the flow control that would prevent a user from sending data over that stream.

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

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

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

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

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

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

    if ((pollfds[0].fd =
            open("/dev/comm01", O_RDWR|O_NDELAY)) < 0) {
        perror("open failed for /dev/comm01");
        exit(1);
    }
    if ((pollfds[1].fd =
            open("/dev/comm02", O_RDWR|O_NDELAY)) < 0) {
        perror("open failed for /dev/comm02");
        exit(2);
    }
```

The variable *pollfds* is declared as an array of `pollfd` structures, where this structure is defined in `<poll.h>` and has the following format:

```
struct pollfd {
    int      fd;          /* File descriptor */
    short    events;      /* Requested events */
    short    revents;     /* Returned events */
}
```

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

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

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

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

```
                perror("error event");
                exit(4);
            case 0:                         /* No events */
                break;
            case POLLIN:
                /* Echo incoming data on "other" Stream */
                while ((count =
                    read(pollfds[i].fd, buf, 1024)) > 0)
                    /*
                     * write loses data if flow control
                     * prevents transmit at this time.
                     */
                    if (write((i == 0 ?
                            pollfds[1].fd: pollfds[0].fd),
                            buf, count) != count)
                        fprintf(stderr,"write lost data\n");
                break;
            }
        }
    }
}
```

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

The second argument to `poll()` specifies the number of entries in the *pollfds* array (2 in this example). The third argument is a timeout value indicating the number of milliseconds `poll()` should wait for an event if none has occurred. On a system where millisecond accuracy is not available, *timeout* is rounded up to the nearest legal value available on that system. Here, the value of *timeout* is -1, specifying that `poll()` should block indefinitely until a requested event occurs or until the call is interrupted.

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

If *revents* is set to a value other than 0 or `POLLIN`, an error event must have occurred on that stream, because the only requested event was `POLLIN`. The following error events are defined for `poll()`. These events may not be polled for by the user, but will be reported in *revents* whenever they occur. As such, they are only valid in the *revents* bitmask:

`POLLERR`
: A fatal error has occurred in some module or driver on the stream associated with the specified file descriptor. Further system calls will fail.

`POLLHUP`
: A hangup condition exists on the stream associated with the specified file descriptor.

`POLLNVAL`
The specified file descriptor is not associated with an open stream.

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

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

Asynchronous Input/Output

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

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

The I_SETSIG `ioctl()` call (see the `streamio`(4) man page) is used to request that a SIGPOLL signal be sent to a user process when a specific event occurs. Listed below are the events for which an application may be signaled:

`S_INPUT`
Data has arrived at the stream head, and no data existed at the stream head when it arrived.

`S_HIPRI`
A priority STREAMS message has arrived at the stream head.

`S_OUTPUT`
The stream is no longer full and can accept output. That is, the stream has relieved the flow control that would prevent a user from sending data over that stream.

`S_MSG`
A special STREAMS signal message that contains a SIGPOLL signal has reached the front of the stream head input queue. This message may be sent by modules or drivers to generate immediate notification of data or events to follow.

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

Clone Open

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

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

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

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

10.4. Multiplexed Streams

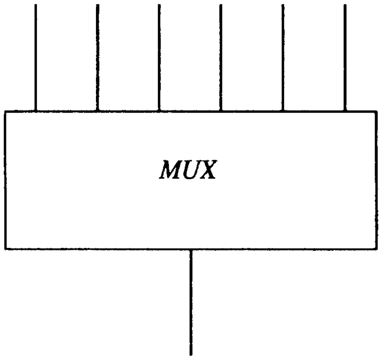
This section describes the construction of multiplexed stream configurations.

Multiplexor Configurations

In the earlier sections, streams were described as linear connections of modules, where each invocation of a module is connected to at most one upstream module and one downstream module. While this configuration is suitable for many applications, others require the ability to multiplex streams in a variety of configurations. Typical examples are terminal window facilities, and internetworking protocols (which might route data over several subnetworks).

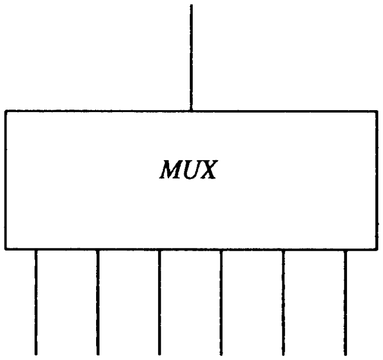
An example of a multiplexor is one that multiplexes data from several upper streams over a single lower stream, as shown in the figure below. An upper stream is one that is upstream from a multiplexor, and a lower stream is one that is downstream from a multiplexor. A terminal windowing facility might be implemented in this fashion, where each upper stream is associated with a separate window.

Figure 10-4 *Many-to-one Multiplexor*



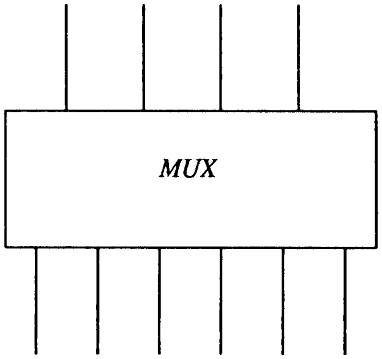
A second type of multiplexor might route data from a single upper stream to one of several lower STREAMS, as shown in the figure below. An internetworking protocol could take this form, where each lower stream links the protocol to a different physical network.

Figure 10-5 *One-to-many Multiplexor*



A third type of multiplexor might route data from one of many upper streams to one of many lower streams, as shown in the figure below.

Figure 10-6 *Many-to-many Multiplexor*



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

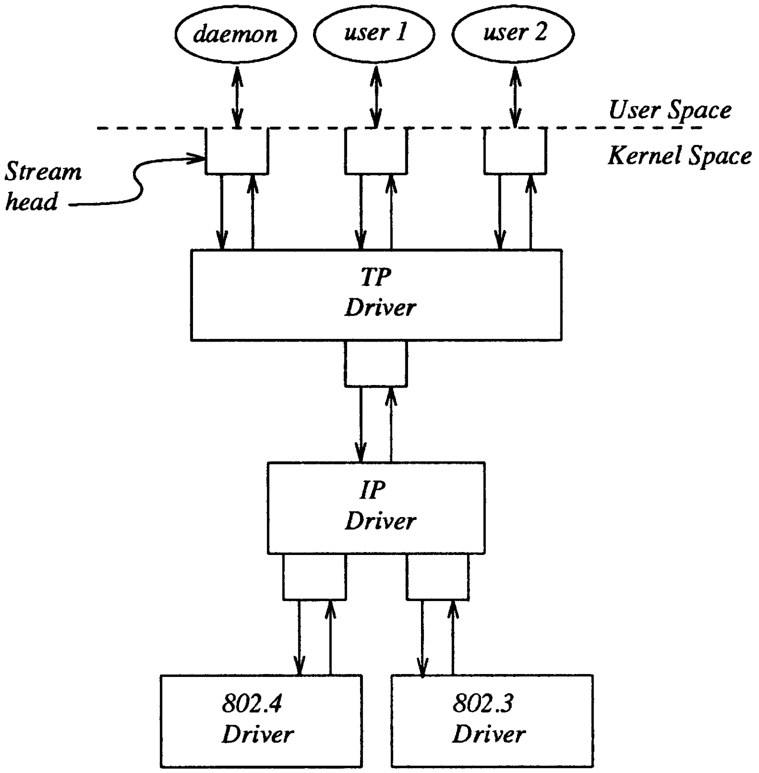
The remainder of this section describes multiplexed stream configurations in the context of an example (see figure below). In this example, an internetworking protocol pseudo-device driver (IP) is used to route data from a single upper stream to one of two lower streams. This driver supports two STREAMS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

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

Building a Multiplexor

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

Figure 10-7 *Protocol Multiplexor*



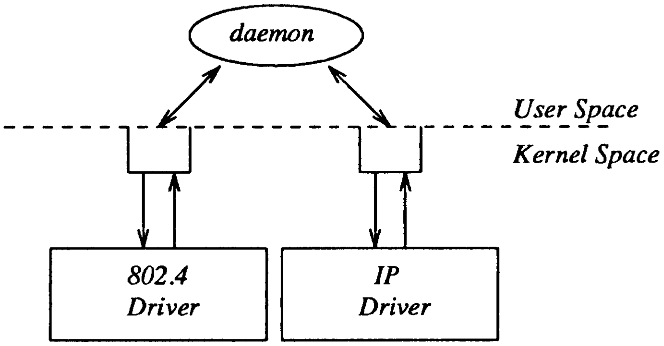
The following example shows how this daemon process sets up the protocol multiplexor. The necessary declarations and initialization for the daemon program are as follows:

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

main()
{
    int fd_802_4,
        fd_802_3,
        fd_ip,
        fd_tp;
    /* Daemon-ize this process */
    switch (fork( )) {
    case 0:
        break;
    case -1:
        perror("fork failed");
        exit(2);
    default:
        exit(0);
    }
    setpgrp( );
```

This multi-level multiplexed stream configuration will be built from the bottom up. Therefore, the example begins by constructing the IP multiplexor. This multiplexing pseudo-device driver is treated like any other software driver. It owns a node in the file system and is opened just like any other STREAMS device driver.

The first step is to open the multiplexing driver and the 802.4 driver, creating separate streams above each driver as shown in the figure below. The stream to the 802.4 driver may now be connected below the multiplexing IP driver using the I_LINK `ioctl()` call.

Figure 10-8 *Before Link*

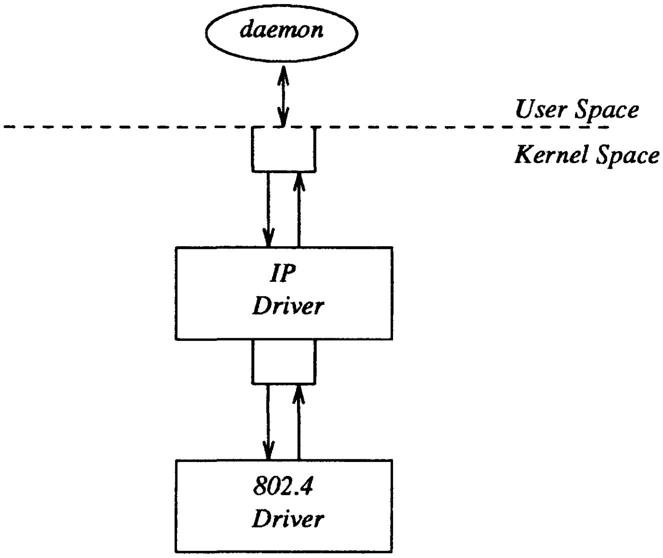


The sequence of instructions to this point is:

```
if ((fd_802_4 = open("/dev/802_4", O_RDWR)) < 0) {
    perror("open of /dev/802_4 failed");
    exit(1);
}
if ((fd_ip = open("/dev/ip", O_RDWR)) < 0) {
    perror("open of /dev/ip failed");
    exit(2);
}
/* Now link 802.4 to underside of IP */
if (ioctl(fd_ip, I_LINK, fd_802_4) < 0) {
    perror("I_LINK ioctl failed");
    exit(3);
}
```

I_LINK takes two file descriptors as arguments. The first file descriptor, *fd_ip*, must reference the stream connected to the multiplexing driver, and the second file descriptor, *fd_802_4*, must reference the stream to be connected below the multiplexor. The figure below shows the state of these streams following the I_LINK call. The complete stream to the 802.4 driver has been connected below the IP driver, including the stream head. The stream head of the 802.4 driver will be used by the IP driver to manage the multiplexor.

Figure 10-9 *IP Multiplexor After First Link*



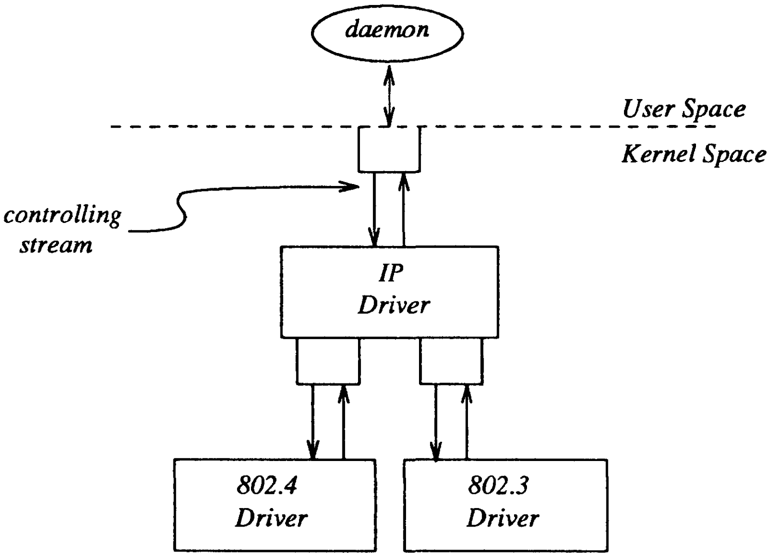
I_LINK will return an integer value, called a mux id, which is used by the multiplexing driver to identify the stream just connected below it. This mux id is ignored in the example, but may be useful for dismantling a multiplexor or routing data through the multiplexor. Its significance is discussed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

The following sequence of system calls is used to continue building the internetworking multiplexor (IP):

```
if ((fd_802_3 = open("/dev/802_3", O_RDWR)) < 0) {
    perror("open of /dev/802_3 failed");
    exit(4);
}
if (ioctl(fd_ip, I_LINK, fd_802_3) < 0) {
    perror("I_LINK ioctl failed");
    exit(5);
}
```

All links below the IP driver have now been established, giving the configuration in the figure below.

Figure 10-10 *IP Multiplexor*



The stream above the multiplexing driver used to establish the lower connections is the controlling stream and has special significance when dismantling the multiplexing configuration, as will be illustrated later in this section. The stream referenced by *fd_ip* is the controlling stream for the IP multiplexor.

NOTE *The order in which the streams in the multiplexing configuration are opened is unimportant. If, however, it is necessary to have intermediate modules in the stream between the IP driver and media drivers, these modules must be added to the streams associated with the media drivers (using I_PUSH) before the media drivers are attached below the multiplexor.*

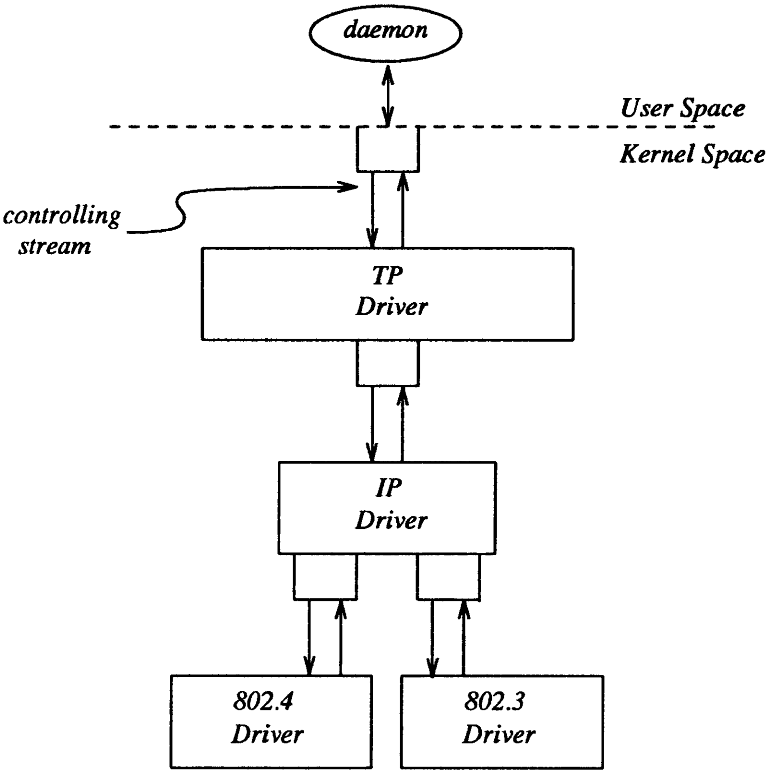
The number of streams that can be linked to a multiplexor is restricted by the design of the particular multiplexor. The manual page describing each driver (typically found in section 7 of the *SunOS Reference Manual*) should describe such restrictions. However, only one I_LINK operation is allowed for each lower stream; a single stream cannot be linked below two multiplexors simultaneously.

Continuing with the example, the IP driver will now be linked below the transport protocol (TP) multiplexing driver. As seen earlier in the figure below, only one link will be supported below the transport driver. This link is formed by the following sequence of system calls:

```
if ((fd_tp = open("/dev/tp", O_RDWR)) < 0) {
    perror("open of /dev/tp failed");
    exit(6);
}
if (ioctl(fd_tp, I_LINK, fd_ip) < 0) {
    perror("I_LINK ioctl failed");
    exit(7);
}
```

The multi-level multiplexing configuration shown in the figure below has now been created.

Figure 10-11 *TP Multiplexor*



Because the controlling stream of the IP multiplexor has been linked below the TP multiplexor, the controlling stream for the new multi-level multiplexor configuration is the stream above the TP multiplexor.

At this point the file descriptors associated with the lower drivers can be closed without affecting the operation of the multiplexor. Closing these file descriptors may be necessary when building large multiplexors, so that many devices can be linked together without exceeding the system limit on the number of simultaneously open files per process. If these file descriptors are not closed, all subsequent `read()`, `write()`, `ioctl()`, `poll()`, `getmsg()`, and `putmsg()` system calls issued to them will fail. That is because I_LINK associates the stream head of each linked stream with the multiplexor, so the user may not access that stream directly for the duration of the link.

The following sequence of system calls will complete the multiplexing daemon example:

```
        close(fd_802_4);
        close(fd_802_3);
        close(fd_ip);
        /* Hold multiplexor open forever */
        pause();
}
```

The figure below shows the complete picture of the multi-level protocol multiplexor. The transport driver is designed to support several, simultaneous virtual circuits, where these virtual circuits map one-to-one to streams opened to the transport driver. These streams will be multiplexed over the single stream connected to the IP multiplexor. The mechanism for establishing multiple streams above the transport multiplexor is actually a by-product of the way in which streams are created between a user process and a driver. By opening different minor devices of a STREAMS driver, separate streams will be connected to that driver. Of course, the driver must be designed with the intelligence to route data from the single lower stream to the appropriate upper stream.

Notice in the figure below that the daemon process maintains the multiplexed stream configuration through an open stream (the controlling stream) to the transport driver. Meanwhile, other users can access the services of the transport protocol by opening new streams to the transport driver; they are freed from the need for any unnecessary knowledge of the underlying protocol configurations and sub-networks that support the transport service.

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

Dismantling a Multiplexor

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

The I_UNLINK `ioctl()` call is used to disconnect each multiplexor link below a multiplexing driver individually. This command takes the following form:

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

where *fd* is a file descriptor associated with a stream connected to the multiplexing driver from above, and *mux_id* is the identifier that was returned by I_LINK when a driver was linked below the multiplexor. Each lower driver may be disconnected individually in this way, or a special *mux_id* value of -1 may be used to disconnect all drivers from the multiplexor simultaneously.

In the multiplexing daemon program presented earlier, the multiplexor is never explicitly dismantled. That is because all links associated with a multiplexing driver are automatically dismantled when the controlling stream associated with that multiplexor is closed. Because the controlling stream is open to a driver, only the final call of `close()` for that stream will close it. In this case, the daemon is the only process that has opened the controlling stream, so the multiplexing configuration will be dismantled when the daemon exits.

For the automatic dismantling mechanism to work in the multi-level, multiplexed stream configuration, the controlling stream for each multiplexor at each level must be linked under the next higher level multiplexor. In the example, the controlling stream for the IP driver was linked under the TP driver. This resulted in a single controlling stream for the full, multi-level configuration. Because the multiplexing program relied on closing the controlling stream to dismantle the multiplexed stream configuration instead of using explicit I_UNLINK calls, the mux id values returned by I_LINK could be ignored.

An important side effect of automatic dismantling on `close()` is that it is not possible for a process to build a multiplexing configuration and then exit. That is because `exit`(2) will close all files associated with the process, including the controlling stream. To keep the configuration intact, the process must exist for the life of that multiplexor. That is the motivation for implementing the example as a daemon process.

Routing Data Through a Multiplexor

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

One routing option available to the multiplexor is to use the mux id value to determine to which stream data should be routed (remember that each multiplexor link is associated with a mux id). I_LINK passes the mux id value to the driver and returns this value to the user. The driver can therefore specify that the mux id value must accompany data routed through it. For example, if a multiplexor routed data from a single upper stream to one of several lower streams (as did the IP driver), the multiplexor could require the user to insert the mux id of the desired lower stream into the first four bytes of each message passed to it. The driver could then match the mux id in each message with the mux id of each lower stream, and route the data accordingly.

10.5. Message Handling

This section describes how to process STREAMS messages in a service interface.

Service Interface Messages

A STREAMS message format has been defined to simplify the design of service interfaces. Also, two new system calls, `getmsg(2)` and `putmsg(2)` are available for sending these messages downstream and receiving messages that are available at the stream head. This section describes these system calls in the context of a service interface example. First, a brief overview of STREAMS service interfaces is presented.

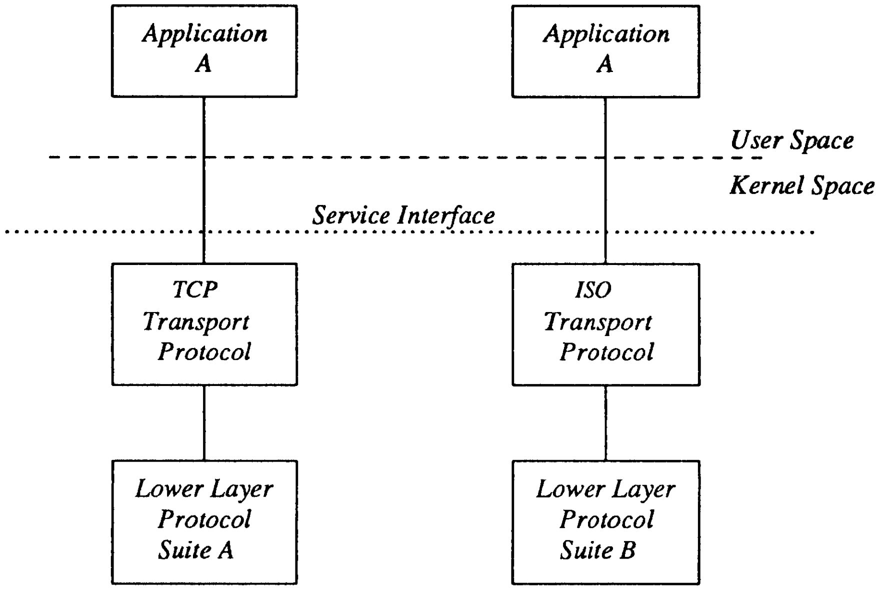
Service Interfaces

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

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

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

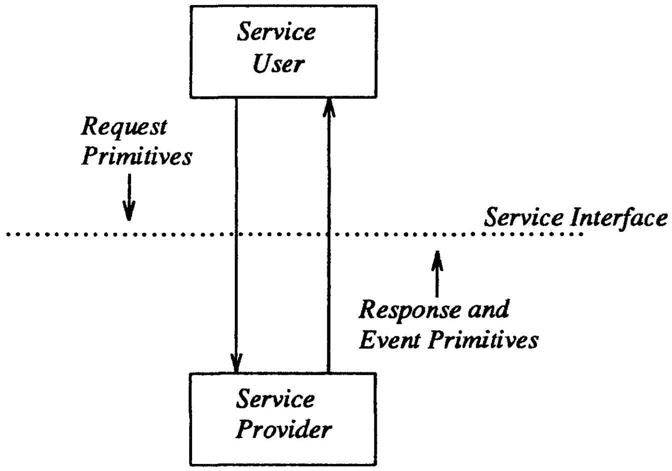
Figure 10-12 *Protocol Substitution*



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

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

Figure 10-13 *Service Interface*



Typically, a user makes a request of a service provider using some well-defined service primitive. Responses and event indications are also passed from the provider to the user using service primitives. The service interface is defined as the set of primitives that define a service and the allowable state transitions that result as these primitives are passed between the user and provider.

The Message Interface

A message format has been defined to simplify the design of service interfaces using STREAMS. Each service interface primitive is a distinct STREAMS message that has two parts: a control part and a data part. The control part contains information that identifies the primitive and includes all necessary parameters. The data part contains user data associated with that primitive.

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

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

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

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

The syntax of `putmsg()` is as follows:

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

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

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

```
struct strbuf {
    int maxlen;         /* Maximum buffer length */
    int len;            /* Length of data */
    char    *buf;       /* Pointer to buffer */
}
```

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

The `getmsg()` system call retrieves messages available at the stream head, and has the following syntax:

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

The arguments to `getmsg()` are the same as those for `putmsg()`.

The remainder of this section presents an example that demonstrates how `putmsg()` and `getmsg()` may be used to interact with the service interface of a simple datagram protocol provider. A potential provider of such a service might be the IEEE 802.2 Logical Link Control Protocol Type 1. The example implements a user level library that would free the user from knowledge of the underlying STREAMS system calls. The Transport Interface of the Network Services Library in UNIX System V Release 3.0 provides a similar function for transport layer services. The example here illustrates how a service interface might be defined, and is not an example of a complete IEEE 802.2 service interface.

Datagram Service Interface Example

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

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

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

```
/*
 * Primitives initiated by the service user.
 */
#define BIND_REQ        1    /* Bind request */
#define UNITDATA_REQ    2    /* Unitdata request */
/*
 * Primitives initiated by the service provider.
 */
#define OK_ACK          3    /* Bind acknowledgment */
#define ERROR_ACK       4    /* Error acknowledgment */
#define UNITDATA_IND    5    /* Unitdata indication */
/*
 * The following structure definitions define the format
 * of the control part of the service interface message
 * of the above primitives.
 */
struct bind_req {            /* Bind request */
    long    PRIM_type;       /* Always BIND_REQ */
    long    BIND_addr;       /* Addr to bind */
};
struct unitdata_req {        /* Unitdata request */
    long    PRIM_type;       /* Always UNITDATA_REQ */
    long    DEST_addr;       /* Destination addr */
};
struct ok_ack {            /* Positive acknowledgment */
    long    PRIM_type;       /* Always OK_ACK */
};
struct error_ack {           /* Error acknowledgment */
    long    PRIM_type;       /* Always ERROR_ACK */
    long    UNIX_error;      /* UNIX error code  */
};
struct unitdata_ind {        /* Unitdata indication */
    long    PRIM_type;       /* Always UNITDATA_IND */
    long    SRC_addr;            /* Source addr */
};
union primitives {         /* Union of all primitives */
    long                     type;
    struct bind_req          bind_req;
    struct unitdata_req      unitdata_req;
    struct ok_ack            ok_ack;
    struct error_ack         error_ack;
    struct unitdata_ind      unitdata_ind;
};
/* Header files needed by library */
#include <stropts.h>
#include <stdio.h>
#include <errno.h>
```

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

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

`UNITDATA_REQ`
This request asks the provider to send a datagram to the specified destination address. It does not require an acknowledgement from the provider.

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

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

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

`UNITDATA_IND`
This primitive indicates that a datagram destined for the user has arrived.

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

Accessing the Datagram Provider

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

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

    if ((fd = open(path, oflags)) < 0)
        return(-1);
    /* Send bind request msg down stream */
    bind_req.PRIM_type = BIND_REQ;
    bind_req.BIND_addr = addr;
    ctlbuf.len = sizeof(struct bind_req);
    ctlbuf.buf = (char *)&bind_req;
    if (putmsg(fd, &ctlbuf, NULL, 0) < 0) {
        close(fd);
        return(-1);
    }
```

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

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

```
    /* Wait for ack of request */
    ctlbuf.maxlen = sizeof(union primitives);
    ctlbuf.len = 0;
    ctlbuf.buf = (char *)&rcvbuf;
    flags = RS_HIPRI;
    if (getmsg(fd, &ctlbuf, NULL, &flags) < 0) {
        close(fd);
        return(-1);
    }
    /* Did we get enough to determine type */
    if (ctlbuf.len < sizeof(long)) {
        close(fd);
        errno = EPROTO;
        return(-1);
    }
    /* Switch on type (first long in rcvbuf) */
    switch(rcvbuf.type) {
        default:
            errno = EPROTO;
            close(fd);
            return(-1);
        case OK_ACK:
            return(fd);
        case ERROR_ACK:
            if (ctlbuf.len < sizeof(struct error_ack)) {
                errno = EPROTO;
                close(fd);
                return(-1);
            }
            error_ack = (struct error_ack *)&rcvbuf;
            errno = error_ack->UNIX_error;
            close(fd);
            return(-1);
    }
}
```

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

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

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

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

Because neither acknowledgement primitive contains a data part, the *dataptr* argument to `getmsg()` is set to NULL. The *flags* argument points to an integer containing the value `RS_HIPRI`. This flag indicates that `getmsg()` should wait for a STREAMS priority message before returning, and is set because the acknowledgement primitives are priority messages. Even if a normal message is available, `getmsg()` will block until a priority message arrives.

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

Closing the Service

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

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

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

Sending a Datagram

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

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

    unitdata_req.PRIM_type = UNITDATA_REQ;
    unitdata_req.DEST_addr = addr;
    ctlbuf.len = sizeof(struct unitdata_req);
    ctlbuf.buf = (char *)&unitdata_req;
    databuf.len = len;
    databuf.buf = buf;
    if (putmsg(fd, &ctlbuf, &databuf, 0) < 0)
        return(-1);
    return(len);
}
```

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

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

Receiving a Datagram

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

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

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

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

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

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

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

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

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

11

STREAMS Module and Driver Programming

STREAMS Module and Driver Programming **237**

11.1. Introduction 237
  Development Facilities 238
11.2. Streams Mechanism 238
  Stream Construction 239
  Opening a Stream 241
  Adding and Removing Modules 242
  Closing 242
11.3. Modules 243
  Module Declarations 243
  Module Procedures 245
  Module and Driver Environment 246
11.4. Messages 247
  Message Format 247
  Filter Module Declarations 249
  Message Allocation 251
  Put Procedure 251
11.5. Message Queues and Service Procedures 253
  The **`queue_t`** Structure 253
  Service Procedures 254
  Message Queues and Message Priority 254
  Flow Control 255
  Example 256

11.6. Drivers 259

  Overview of Drivers 259

  Driver Flow Control 261

  Driver Programming 262

  Driver Processing Procedures 265

  Driver and Module Ioctls 267

  Driver Close 269

11.7. Complete Driver 269

  Cloning 269

  Loop-Around Driver 270

11.8. Multiplexing 278

  Multiplexing Configurations 278

  Multiplexor Construction Example 281

  Multiplexing Driver 284

11.9. Service Interface 294

  Definition 294

  Example 295

11.10. Advanced Topics 299

  Recovering From No Buffers 299

  Advanced Flow Control 301

  Signals 302

  Control of Stream Head Processing 303

11

STREAMS Module and Driver Programming

11.1. Introduction

This chapter provides detailed information on the use of the STREAMS mechanism at the kernel level, including examples, information on development methods and design philosophy. It describes the use of STREAMS kernel facilities for developing and installing modules and drivers, and is intended for system programmers with knowledge of UNIX kernel programming, device driver development, networking and other data communication facilities.

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

The preceding *STREAMS Application Programming* chapter is the companion to this chapter—it provides an analogous discussion of the STREAMS applications level.

Both of these chapters assumes a working knowledge of the material covered in the preceding *Introduction to STREAMS* chapter (hereafter simply called the *Introduction to STREAMS*). This introduction includes a useful `Glossary` of STREAMS-related terms. STREAMS kernel utilities are summarized in the *Utilities* section of the *Supplementary STREAMS Material* chapter of this manual. STREAMS system calls are specified in Section 2 of the *SunOS Reference Manual.* The STREAMS modules and drivers available with SunOS are described in section 4 of the *SunOS Reference Manual.* STREAMS-specific `ioctl()` calls are specified in `streamio(4)`.

STREAMS was incorporated into SunOS to augment the existing character input/output (I/O) mechanism and to support the development of communication services. A STREAMS driver may be a device driver that provides the services of an external I/O device, or a software driver, commonly referred to as a pseudo-device driver, that performs functions internal to a Stream. The Stream head provides the interface between the Stream and user processes. Its principal function is to process STREAMS-related user system calls so that this processing does not have to be incorporated in a module and driver.

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

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

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

Development Facilities

General and STREAMS-specific system calls provide the user level facilities required to implement application programs. This system call interface is upwardly compatible with the character I/O facilities. The `open(2)` system call recognizes a STREAMS file and creates a Stream to the specified driver. A user process can receive and send data on STREAMS files using `read(2)` and `write(2)` in the same manner as with character files. The `ioctl(2)` system call enables users to perform functions specific to a particular device and a set of generic STREAMS `ioctl()` commands (see `streamio`(4)) support a variety of functions for accessing and controlling Streams. A `close(2)` will dismantle a Stream.

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

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

- Message storage management – to maintain STREAMS' own memory resources for message storage
- Flow control – to conserve STREAMS memory and processing resources
- Scheduling – to control the execution of service procedures
- Multiplexing – to switch data among multiple Streams

11.2. Streams Mechanism

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

The components that form a Stream are the Stream head, driver and optional modules. A Stream is initially constructed as the result of a user process `open(2)` system call referencing a STREAMS file. The call causes a kernel resident driver to be connected with a Stream head to form a Stream. Subsequent `ioctl(2)` calls select kernel resident modules and cause them to be inserted in the Stream. A module represents intermediate processing on messages flowing between the Stream head and driver. A module can function as, for example, a communication protocol, line discipline or data filter. STREAMS allows a user to connect a module with any other module. The user determines the module connection sequences that result in useful configurations.

A process can send and receive characters on a Stream using `write(2)` and `read(2)`, as on character files. When user data enters the Stream head or external data enters the driver, the data is placed into messages for transmission on the Stream. All data passed on a Stream is carried in messages, each having a defined message type identifying the message contents. Internal control and status information is transmitted among modules or between the Stream and user process as messages of certain types interleaved on the Stream. Modules and drivers can send certain message types to the Stream head to cause the generation of signals or errors to be received by the user process.

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

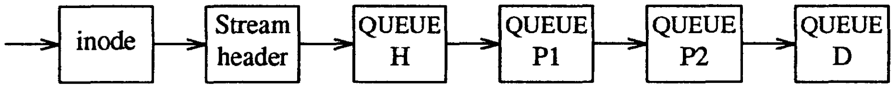
The last `close(2)` system call dismantles the Stream and closes the file, semantically identical to character I/O drivers.

STREAMS supports implementation of user level applications with extensions to the above general system calls and STREAMS specific system calls: `putmsg(2)`, `getmsg(2)`, `poll(2)` and a set of STREAMS generic `ioctl(2)` functions.

Stream Construction

STREAMS constructs a Stream as a linked list of kernel resident data structures. In a STREAMS file, the `vnode` points to the Stream `header` structure. The `header` is used by STREAMS kernel routines to perform operations on this Stream generally related to system calls. Figure *11-1* depicts the downstream (write) portion of a Stream (see *Building a Stream*, in the *Introduction to STREAMS*) connected to a `header`. There is one `header` per Stream. From the `header` onward, a Stream is constructed of QUEUEs. The upstream (read) portion of the Stream (not shown here) parallels the downstream portion in the opposite direction and terminates at the Stream `header` structure.

Figure 11-1 *Downstream Stream Construction*


inode
Stream header
QUEUE H
QUEUE P1
QUEUE P2
QUEUE D

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

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

Figure *11-2* shows the data structures forming each QUEUE: `queue_t`, `qinit`, `module_info` and `module_stat`. `queue_t` contains various modifiable values for this QUEUE, generally used by STREAMS. `qinit` contains a pointer to the processing procedures, `module_info` contains limit values and `module_stat` is used for statistics. The two QUEUEs in a module will generally each contain a different set of these structures. The contents of these structures are described in following sections.

Figure 11-2 *QUEUE data structures*

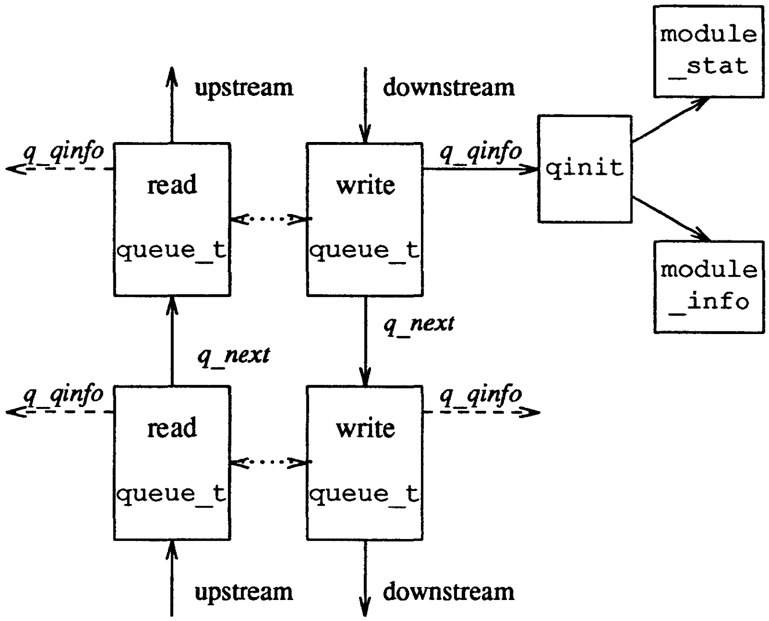

module _stat
upstream
downstream
q_qinfo
read queue_t
write queue_t
q_qinfo
qinit
module _info
q_next
q_next
q_qinfo
read queue_t
write queue_t
q_qinfo
upstream
downstream

Figure *11-1* shows QUEUE linkage in one direction while figure *11-2* shows two neighboring modules with links (solid vertical arrows) in both directions. When a module is pushed onto a Stream, STREAMS creates two QUEUEs and links each QUEUE in the module to its neighboring QUEUE in the upstream and downstream direction. The linkage allows each QUEUE to locate its next neighbor. The next relation is implemented between `queue_ts` in adjacent modules by the `q_next` pointer. Within a module, each `queue_t` locates its mate (see dotted arrows in figure *11-2*) by use of STREAMS macros, since there is no pointer between the two `queue_ts`. The existence of the Stream head and driver is known to the QUEUE procedures only as destinations towards which messages are sent.

Opening a Stream

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

```
struct streamtab {
    struct qinit    *st_rdinit;    /* defines read QUEUE */
    struct qinit    *st_wrinit;    /* defines write QUEUE */
    struct qinit    *st_muxrinit;  /* for multiplexing drivers only */
    struct qinit    *st_muxwinit;  /* for multiplexing drivers only */
    char            **st_modlist;  /* list of modules to be pushed */
}
```

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

If this `open()` call is the initial file open, a Stream is created. First, the single `header` structure and the Stream head (see figure *11-1*) `queue_t` structure pair are allocated. Their contents are initialized with predetermined values including, as noted above (see QUEUE H), the Stream head processing routines.

Then, a `queue_t` structure pair is allocated for the driver. The `queue_t` contents are zero unless specifically initialized (see the *Message Queues and Service Procedures* section). A single, common `qinit` structure pair is shared among all the Streams opened from the same `cdevsw` entry, as is the associated `module_info` and `module_stat` structures (see figure *11-2*.)

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

If the `st_modlist` pointer is not `NULL`, it is assumed to point to the first member of an array of pointers to module names. After the driver's `open()` procedure has been called, the modules whose names are pointed to be the members of that array are pushed onto the stream, in the order that they appear in the array. (See *Adding and Removing Modules*, below). If one of these modules cannot be pushed, the `open()` fails.

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

Adding and Removing Modules

As part of constructing a Stream, a module can be added with an `ioctl()` I_PUSH (see the `streamio(4)` man page) system call (push). The push inserts a module beneath the Stream head. Because of the similarity of STREAMS components, the push operation is similar to the driver open. First, the address of the `qinit` structure for the module is obtained via an `fmodsw` entry.

`fmodsw` is an array, analogous to `cdevsw`. Each `fmodsw` entry corresponds to a unique module and contains the name of the module (used by I_PUSH and certain other STREAMS `ioctl()`s) and a pointer to the module's `streamtab`. Next, STREAMS allocates `queue_t` structures and initializes their contents as in the driver open, above. As with the driver, the read and write `qinit` structures are shared among all the modules opened from this `fmodsw` entry (see figure *11-2.*)

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

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

An `ioctl()` I_POP (see the `streamio(4)` man page) system call (pop) removes the module immediately below the Stream head. The pop calls the module close procedure. On return from the module close, any messages left on the module's message queues are freed (deallocated). Then, STREAMS connects the Stream head to the component previously below the popped module and deallocates the module's two `queue_t` structures. I_POP enables a user process to dynamically alter the configuration of a Stream by pushing and popping modules as required. For example, a module may be removed or a new one inserted below a module. In the latter case, the original module is popped and pushed back after the new module has been pushed.

An I_POP cannot be used on a driver.

Closing

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

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

Finally, the file is closed.

11.3. Modules

Module Declarations

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

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

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

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

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

The declarations shown are: the header set; the read and write QUEUE (`rminfo` and `wminfo`) `module_info` structures (see figure *11-2*); the module open, read-put, write-put and close procedures; the read and write (`rinit` and `winit`) `qinit` structures; and the `streamtab` structure.

The minimum header set for modules and drivers is `types.h` and `stream.h`. `param.h` contains definitions for NULL and other values for STREAMS modules and drivers as shown in the *Accessible Symbols and Functions* section of the *Supplementary STREAMS Material* chapter.

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

As described in the previous section, `streamtab` contains `qinit` values for the read and write QUEUEs, pointing to a `module_info` and an optional `module_stat` structure. The two required structures, shown in figure *11-2*), are these:

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

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

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

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

Three names are associated with a module: the character string in `fmodsw`; the prefix for `streamtab`, used in configuring the module; and the module name field in the `module_info` structure. The module name value used in the I_PUSH or other STREAMS `ioctl()` commands is contained in `fmodsw`. Each module ID and module name should be unique in the system. The module ID is currently used only in logging and tracing (see *Other Facilities*, in the *Introduction to STREAMS*). For the example in this section, the module ID is zero.

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

`module_stat` is optional, intended for future use. Currently, there is no STREAMS support for statistical information gathering. The structure is described in *Kernel Structures* in the *Supplementary STREAMS Material* chapter.

Module Procedures

The null module procedures are as follows:

```
static int modopen(q, dev, flag, sflag)
    queue_t *q;        /* pointer to read queue */
    dev_t dev;         /* major/minor device number -- zero for modules */
    int  flag;         /* file open flags -- zero for modules */
    int  sflag;        /* stream open flags */
{
    /* return success */
    return 0;
}

static int modwput(q, mp)          /* write put procedure */
    queue_t *q;        /* pointer to the write queue */
    mblk_t  *mp;       /* message pointer */
{
    putnext(q, mp);  /* pass message through */
}

static int modrput(q, mp)          /* read put procedure */
    queue_t *q;        /* pointer to the read queue */
    mblk_t  *mp;       /* message pointer */
{
    putnext(q, mp);  /* pass message through */
}

static int modclose(q, flag)
    queue_t *q;        /* pointer to the read queue */
    int  flag;         /* file open flags - zero for modules */
{
}
```

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

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

LBMODOPEN
normal module open

LBO
normal driver open (see the *Drivers* section).

LBCLONEOPEN
clone driver open (see the *Complete Driver* section).

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

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

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

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

Module and Driver Environment

User context is not generally available to STREAMS module procedures and drivers. The exception is during execution of the open and close routines. Driver and module open and close routines have user context and may access the `user` structure (defined in `user.h`, see *Accessible Symbols and Functions* in the *Supplementary STREAMS Material*) chapter. These routines are allowed to sleep, but must always return to the caller. That is, if they sleep, it must be at priority <= PZERO, or with PCATCH set in the sleep priority. (A process which is sleeping at priority > PZERO and is sent a signal via `kill(2)`, never returns from the sleep call. Instead, the system call is aborted.)

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

11.4. Messages

Message Format

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

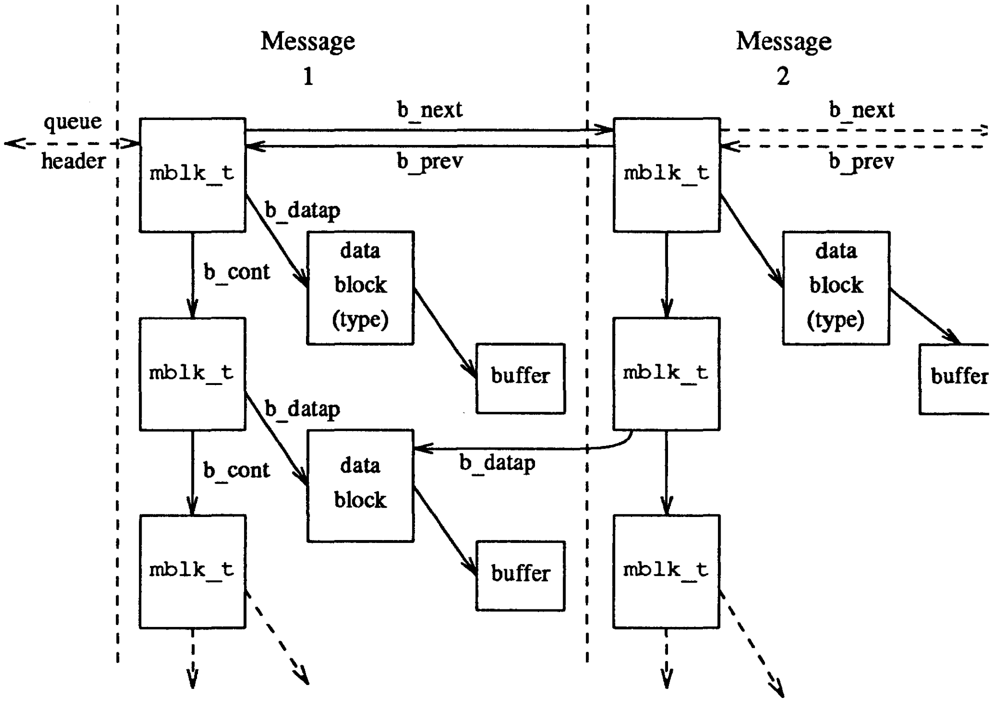
All messages are composed of one or more message blocks. A message block is a linked triplet, two structures and a variable length buffer block. The structures are `msgb` (`mblk_t`), the message block, and `datab` (`dblk_t`), the data block:

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

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

`mblk_t` is used to link messages on a message queue, link the blocks in a message and manage the reading and writing of the associated buffer. `b_rptr` and `b_wptr` are used to locate the data currently contained in the buffer. As shown in figure *11-3*, `mblk_t` points to the data block of the triplet. The data block contains the message type, buffer limits and control variables. STREAMS allocates message buffer blocks of varying sizes (see below). `db_base` and `db_lim` are the fixed beginning and end (+1) of the buffer.

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

Figure 11-3 *Message Form and Linkage*


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

A message may occur singly, as when it is processed by a put procedure, or it may be linked on the message queue in a QUEUE, generally waiting to be processed by the service procedure. Message 1, as shown in figure *11-3*, links to message 2. In the first message on a queue, `b_prev` points back to the header in the QUEUE. The last `b_next` points to the tail.

Note that a data block in message 1 is shared between message 1 and another message. Multiple message blocks can point to the same data block to conserve storage and to avoid copying overhead. For example, the same data block, with associated buffer, may be referenced in two messages, from separate modules that implement separate protocol levels. (Figure *11-3* illustrates the concept, but data blocks would not typically be shared by messages on the same queue). The buffer can be retransmitted, if required by errors or timeouts, from either protocol level without replicating the data. Data block sharing is accomplished by means of a utility routine (see `dupmsg()` in the *Utilities* section of the *Supplementary STREAMS Material* chapter. STREAMS maintains a count of the message blocks sharing a data block in the `db_ref` field.

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

Message Generation and Reception

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

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

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

Filter Module Declarations

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

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

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

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

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

Note that, in contrast to the null module example, a single `module_info` structure is shared by the read and write sides.

`modopen()`, `modrput()` and `modclose()` are the same as in the null module of the preceding section.

`bappend()` Subroutine

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

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

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

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

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

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

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

Message Allocation

The `allocb()` utility (see the *Utilities* section of the *Supplementary STREAMS Material* chapter) is used to allocate message storage from the STREAMS pool. Its declaration is:

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

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

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

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

Put Procedure

`modwput()` processes all the message blocks in any downstream data (type

M_DATA) messages.

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

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

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

                bp->b_rptr++;
                continue;

            newblk:
                if (nmp == NULL)
                    nmp = nbp;

                /* link message block to tail of nmp */
                else linkb(nmp, nbp);

                nbp = NULL;
            }
        }

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

Data messages are scanned and filtered. `modwput()` copies the original message into a new block(s), modifying as it copies. `nbp` points to the current new message block. `nmp` points to the new message being formed as multiple M_DATA message blocks. The outer for() loop goes through each message block of the original message. The inner while() loop goes through each byte.

`bappend()` is used to add characters to the current or new block. If `bappend()` fails, the current new block is full. If nmp is NULL, nmp is pointed at the new block. If nmp is non-NULL, the new block is linked to the end of nmp by use of the `linkb` utility.

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

11.5. Message Queues and Service Procedures

The `queue_t` Structure

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

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

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

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

- `q_qinfo` - from `streamtab`
- `q_minpsz`, `q_maxpsz`, `q_hiwat`, `q_lowat` - from `module_info` Copying values from `module_info` allows them to be changed in the `queue_t` without modifying the template (i.e., `streamtab` and `module_info`) values.

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

Service Procedures

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

A service routine must never sleep and it has no user context. It must always return to its caller.

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

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

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

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

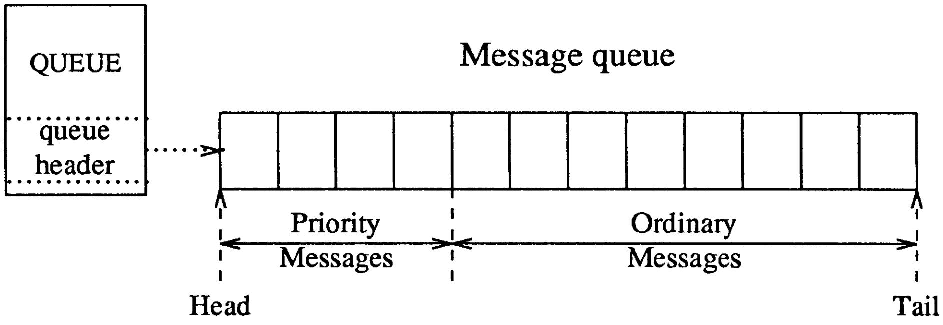
The delay for STREAMS to call a service procedure will vary with implementation and system activity. However, once the service procedure is scheduled, it is guaranteed to be called before user level activity is resumed.

Also see the *Put and Service Procedures* section of the *Introduction to STREAMS*.

Message Queues and Message Priority

Figure *11-3* depicts a message queue linked by `b_next` and `b_prev` pointers. As discussed in the *Introduction to STREAMS*, message queues grow when the STREAMS scheduler is delayed from calling a service procedure because of system activity, or when the procedure is blocked by flow control. When it is called by the scheduler, the service procedure processes enqueued messages in FIFO order. However, certain conditions require that the associated message (e.g., an M_ERROR) reach its Stream destination as rapidly as possible. STREAMS does this by assigning all message types to one of the two levels of message queueing priority—priority and ordinary. As shown in figure *Message Queue Priority*, when a message is queued, the `putq()` utility will place priority messages at the head of the message queue, FIFO within their order of queueing.

Figure 11-4 *Message Queue Priority*



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

The priority level for each message type is shown in the *Message Types* section of the *Supplementary STREAMS Material* chapter. Message queue management utilities are provided for use in service procedures (see the *Utilities* section of the *Supplementary STREAMS Material* chapter).

Flow Control

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

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

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

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

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

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

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

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

When an ordinary message is enqueued by `putq()`, `putq()` will cause the service procedure to be scheduled only if the queue was previously empty. If there are messages on the queue, `putq()` presumes the service procedure is blocked by flow control and the procedure will be automatically rescheduled by STREAMS when the block is removed. If the service procedure cannot complete processing as a result of conditions other than flow control (e.g., no buffers), it must assure it will return later (e.g., by use of `bufcall()`, see the *Advanced Topics* section) or it must discard all messages on queue. If this is not done, STREAMS will never schedule the service procedure to be run unless the QUEUE's put procedure queues a priority message with `putq()`.

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

Example

The filter module example of the *Messages* section is here modified to have a service procedure. The declarations from the example are unchanged except for the following lines (changes are shown in **bold**):

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

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

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

`stropts.h` is generally intended for user level. However, it includes definitions of flush message options common to user level, modules and drivers. `module_info` now includes the flow control high- and low-water marks (512 and 128) for the write QUEUE (even though the same `module_info` is used on the read QUEUE side, the read side has no service procedure so flow control is not used). `qinit` now contains the service procedure pointer. `modopen()`, `modclose()` and `modrput()` (read side put procedure) are unchanged from the *Modules* and *Messages* sections. The `bappend()` subroutine is also unchanged from the *Messages* section.

Procedures

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

```
static int modwput(q, mp)
queue_t *q;
register mblk_t *mp;
{
    if (mp->b_datap->db_type > QPCTL &&
     mp->b_datap->db_type != M_FLUSH)
        putnext(q, mp);
        else
        putq(q, mp);      /* Put it on the queue  */
}

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

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

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

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

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

`modwput()`, the write put procedure, switches on the message type. Priority messages that are not type M_FLUSH are `putnext()` to avoid scheduling. The others are queued for the service procedure. An M_FLUSH message is a request to remove all messages on one or both QUEUEs. It can be processed in the put or service procedure.

`modwsrv()` is the write service procedure. It takes a single argument, a pointer to the write `queue_t`. `modwsrv()` processes only one priority message, M_FLUSH. All other priority messages are passed through. Actually, no other priority messages should reach `modwsrv()`. The check is included to show the canonical form when priority messages are queued by the put procedure.

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

Ordinary messages will be returned to the queue if

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

returns false, indicating the downstream path is blocked.

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

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

                if (!canput(q->q_next)) {
                    putbq(q, mp);
                    return;
                }
                /* Filter data, appending to queue */
                for (; mp != NULL; mp = next) {
                    while (mp->b_rptr < mp->b_wptr) {

                        if (*mp->b_rptr == '\n')
                            if (!bappend(&nbp, '\r'))
                                goto push;
                        if (!bappend(&nbp, *mp->b_rptr))
                            goto push;
                        mp->b_rptr++;
                        continue;

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

The differences in M_DATA processing between this and the previous example relate to the manner in which the new messages are forwarded and flow control. For the purpose of demonstrating alternative means of processing messages, this version creates individual new messages rather than a single message containing multiple message blocks. When a new message block is full, it is immediately forwarded with `putnext()` rather than being linked into a single, large message (as was done in the previous example). This alternative may not be desirable because message boundaries will be altered and because of the additional overhead of handling and scheduling multiple messages.

When the filter processing is performed (following `push()`), flow control is checked (`canput()`) after, rather than before, each new message is forwarded. This is done because there is no provision to hold the new message until the QUEUE becomes unblocked. If the downstream path is blocked, the remaining part of the original message is returned to the queue. Otherwise, processing continues.

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

11.6. Drivers

Overview of Drivers

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

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

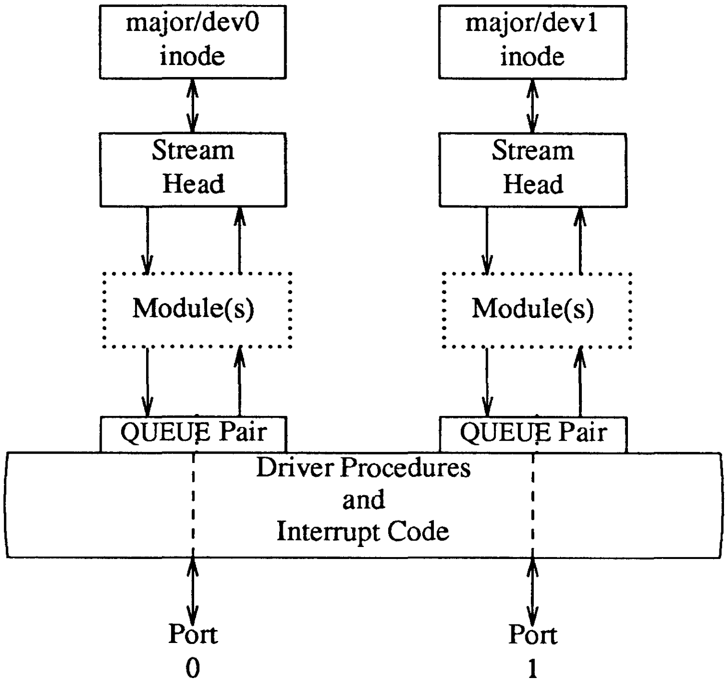
There are two significant differences between modules and drivers. First, a device driver must also be accessible from an interrupt as well as from the Stream, and second, a driver can have multiple Streams connected to it. Multiple connections occur when more than one minor device uses the same driver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

Figure *11-5* shows multiple Streams (corresponding to minor devices), to a common driver. This depiction of two Streams connected to a single driver is somewhat misleading. These are really two distinct Streams opened from the same `cdevsw` (i.e., same major device). Consequently, they have the same `streamtab` and the same driver procedures. Modules opened from the same `fmodsw` might be depicted similarly if they had any reason to be cognizant, as do drivers, of common resources or alternate instantiations.

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

In addition to these differences, a driver is always at the end of a Stream. As a result, drivers must include standard processing for certain message types that a module might simply be able to pass to the next component.

Figure 11-5 *Device Driver Streams*



Driver Flow Control

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

When the message is queued, `putq()` increments the value of `q_count` (approximately the enqueued character count, see the beginning of the *Message Queues and Service Procedures* section) by the size of the message and compares the result against the driver's write high water limit (`q_hiwat`) value. If the count exceeds `q_hiwat`, `putq()` will set the internal FULL (see *Flow Control* in the *Introduction to STREAMS*). indicator for the driver write QUEUE. This will cause messages from upstream to be halted (`canput()` returns FALSE) until the write queue count reaches `q_lowat`. The driver messages waiting to be output are dequeued by the driver output interrupt routine with `getq()`, which decrements the count. If the resulting count is below `q_lowat`, `getq()`

will back-enable any upstream QUEUE that had been blocked. The above STREAMS processing also applies to modules on both write and read sides of the Stream.

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

To some extent, a driver or module can control when its upstream transmission will become blocked. Control is available through the M_SETOPTS message (see the *Advanced Topics* section, here, and the *Message Types* section of the *Supplementary STREAMS Material*) to modify the Stream head read side flow control limits.

Driver Programming

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

Open handling
: A driver is passed a minor device number or is asked to select one (see next section).

Flush handling
: A driver must loop M_FLUSH messages back upstream.

Ioctl handling
: A driver must nak ioctl messages it does not understand. This is discussed under *Driver and Module Ioctls*, below. Write side flow control is also illustrated as described above.

Driver Declarations

The driver declarations are as follows:

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

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

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

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

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

#define SET_OPTIONS (('l'<<8)|1)      /* really must be in a .h file */
/*
 * This is a private data structure, one per minor device number.
 */
struct lp {
    short flags;       /* flags -- see below */
    mblk_t *msg;       /* current message being output */
    queue_t *qptr;     /* back pointer to write queue */
};
/* Flags bits */
#define BUSY  1 /* device is running and interrupt is pending */

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

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

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

`flags`
A set of flags. Only one bit is used: BUSY indicates that output is active and a device interrupt is pending.

`msg`
A pointer to the current message being output.

`qptr`
A back pointer to the write queue. This is needed to find the write queue

during interrupt processing.

Driver Open

The driver open, `lpopen()`, has the same interface as the module open:

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

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

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

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

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

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

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

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

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

`WR` is one of three QUEUE pointer macros. As discussed in the *Stream Construction* section, there are no physical pointers between QUEUEs, and these macros (see *Utilities* in the *Supplementary STREAMS Material* section) generate the pointer. `WR(q)` generates the write pointer from the read pointer, `RD(q)` generates the read pointer from the write pointer and `OTHER`(q) generates the mate pointer from either.

Driver Processing Procedures

This example only has a write put procedure:

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

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

    switch (mp->b_datap->db_type) {
    default:
        freemsg(mp);
        break;
    case M_FLUSH:
        /* Canonical flush handling */
        if (*mp->b_rptr & FLUSHW) {
            flushq(q, FLUSHDATA);
            s = spl5();
            /* also flush lp->msg since it is logically
             * at the head of the write queue */
            if (lp->msg) {
                freemsg(lp->msg);
                lp->msg = NULL;
            }
            splx(s);
        }

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

    case M_IOCTL:
    case M_DATA:
        putq(q, mp);
        s = spl5();
        if (!(lp->flags & BUSY))
            lpout(lp);
        splx(s);
    }
}
```

Driver Flush Handling

The write put procedure, `lpwput()`, illustrates driver M_FLUSH handling: Note that all drivers are expected to incorporate this flush handling. If FLUSHW is set, the write message queue is flushed, and also (for this example) the leading message (`lp->msg`). `spl5` is used to protect the critical code, assuming the device interrupts at level 5. If FLUSHR is set, the read queue is flushed, the FLUSHW bit is cleared, and the message is sent upstream using `qreply()`. If FLUSHR is not set, the message is discarded.

The Stream head always performs the following actions on flush requests received on the read side from downstream. If FLUSHR is set, messages waiting to be sent to user space are flushed. If FLUSHW is set, the Stream head clears the FLUSHR bit and sends the M_FLUSH message downstream. In this manner, a single M_FLUSH message sent from the driver can reach all QUEUEs in a Stream. A module must send two M_FLUSH messages to have the same affect.

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

Driver Interrupt

`lpintr()` is the driver interrupt routine:

```
/* Device interrupt routine.  */

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

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

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

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

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

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

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

`lpout()` simply takes a character from the queue and sends it to the printer. The processing is logically similar to the service procedure in the *Message Queues and Service Procedures* section. For convenience, the message currently being output is stored in p—>`msg`.

Two mythical routines need to be supplied:

lpoutchar
send a character to the printer and interrupt when complete

lpsetopt
set the printer interface options

Driver and Module Ioctls

Drivers and modules interface with `ioctl(2)` system calls through messages. Almost all STREAMS generic `ioctl()`s (see the `streamio(4)` man page) go no further than the Stream head. The capability to send an `ioctl()` downstream, is similar to the `ioctl()` of character device drivers, is provided by the `I_STR ioctl`. The Stream head processes an `I_STR` by constructing an `M_IOCTL` message (see *Message Types* in the *Supplementary STREAMS Material* chapter) from data provided in the call, and sends that message downstream. In addition, since `ioctl()` codes in SunOS include the size of the parameter used for the `ioctl()` as well as an indication of whether this parameter is to be copied to or from the user process, the `I_STR ioctl` need not be used if the parameter contains 255 or fewer bytes as is of a fixed size.

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

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

`lpout()` traps M_IOCTL messages and calls `lpdoioctl()` to process them:

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

    q = lp->qptr;

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

    switch (iocp->ioc_cmd) {
    case SET_OPTIONS:
        /* Count should be exactly one short's worth */
        if (iocp->ioc_count != sizeof(short))
            goto iocnak;
        /* Actual data is in 2nd message block */
        lpsetopt(lp, *(short *)mp->b_cont->b_rptr);

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

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

The form of an `iocblk` is as follows:

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

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

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

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

Driver Close

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

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

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

11.7. Complete Driver

Cloning

The clone mechanism has been developed as a convenience. It allows a user to open a driver without specifying the minor device. When a Stream is opened, a flag indicating a clone open is tested by the driver open routine. If the flag is set, the driver returns an unused minor device number. The clone driver (see the `clone(4)` man page) is a system dependent STREAMS pseudo driver.

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

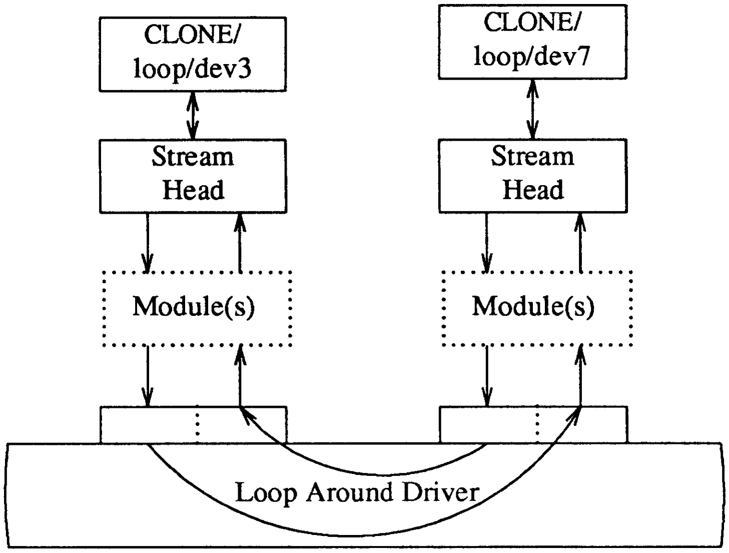
Loop-Around Driver

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

To create a pipe, a process opens two Streams, obtains the minor device number associated with one of the returned file descriptors, and sends the device number in an `ioctl(2)` to the other Stream. For each `open()`, the driver open places the passed `queue_t` pointer in a driver interconnection table, indexed by the device number. When the driver later receives the I_STR as an M_IOCTL message, it uses the device number to locate the other Stream's interconnection table entry, and stores the appropriate `queue_t` pointers in both of the Streams' interconnection table entries.

Subsequently, when messages other than M_IOCTL or M_FLUSH are received by the driver on either Stream's write side, the messages are switched to the read QUEUE following the driver on the other Stream's read side. The resultant logical connection is shown in figure *Loop Around Streams*. Flow control between the two Streams must be handled by special code since STREAMS will not automatically propagate flow control information between two Streams that are not physically interconnected.

Figure 11-6 *Loop Around Streams*


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

The declarations for the driver are:

```
/*
 * Loop around driver
 */

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

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

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

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

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

struct streamtab loopinfo = { &rinit, &winit, NULL, NULL };
struct loop {
    queue_t *qptr;   /* back pointer to write queue */
    queue_t *oqptr;  /* pointer to connected read queue */
};

#define LOOP_SET    _IOW(1, 1, int) /* should be in a .h file */

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

The `loop` structure contains the interconnection information for a pair of Streams. `loop_loop` is indexed by the minor device number. When a Stream is opened to the driver, the address of the corresponding `loop_loop` element is placed in `q_ptr` (private data structure pointer) of the read and write side `queue_ts`. Since STREAMS clears `q_ptr` when the `queue_t` is allocated, a NULL value of `q_ptr` indicates an initial `open()`. `loop_loop` is used to verify that this Stream is connected to another open Stream.

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

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

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

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

    /* Setup data structures */

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

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

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

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

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

Write Put Procedure

Since the messages are switched to the read QUEUE following the other Stream's read side, the driver needs a put procedure only on its write side:

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

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

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

        iocp = (struct iocblk *)mp->b_rptr;
        switch (iocp->ioc_cmd) {
        case LOOP_SET: {
            int to; /* other minor device */
            /*
             * Sanity check. ioc_count contains the amount of
             * user supplied data which must equal the size of an int.
             */

            if (iocp->ioc_count != sizeof(int)) {
                error = EINVAL;
                goto iocnak;
            }

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

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

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

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

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

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

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

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

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

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

            }
            break;

        }
```

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

The driver performs several sanity checks: Does the second block have the proper amount of data? Is the "to" device in range? Is the "to" device open? Is the current Stream disconnected? Is the "to" Stream disconnected?

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

Canonical flush handling is incorporated in the put procedure:

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

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

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

Stream Head Messages

Certain message types (see *Message Types* in the *Supplementary STREAMS Material* chapter) can be sent upstream by drivers and modules to the Stream head where they are translated into actions detectable by user process(es). The messages may also modify the state of the Stream head:

M_ERROR
Causes the Stream head to lock up. Message transmission between Stream and user processes is terminated. All subsequent system calls except `close(2)` and `poll(2)` will fail. Also causes an M_FLUSH clearing all message queues to be sent downstream by the Stream head.

M_HANGUP
Terminates input from a user process to the Stream. All subsequent system calls that would send messages downstream will fail. Once the Stream head read message queue is empty, EOF is returned on reads. Can also result in SIGHUP signal to the process group.

M_SIG/M_PCSIG
Causes a specified signal to be sent to a process (see the *Advanced Topics* section).

Service Procedures

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

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

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

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

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

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

        /* send message */
        /* To queue following other stream read queue */

        putnext(loop->oqptr, mp);
    }
}

static int looprsrv(q)
queue_t *q;
```

```
{

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

    struct loop *loop;

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

    /* manually enable write service procedure */

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

The write service procedure, `loopwsrv()`, takes on the canonical form (see the *Message Queues and Service Procedures* section) with a difference. The QUEUE being written to is not downstream, but upstream (found via `oqptr`) on the other Stream.

In this case, there is no read side put procedure so the read service procedure, `looprsrv()`, is not scheduled by an associated put procedure, as has been done previously. `looprsrv()` is scheduled only by being back-enabled when its upstream becomes unstuck from flow control blockage. The purpose of the procedure is to re-enable the writer (`loopwsrv()`) by using `oqptr` to find the related `queue_t`. `loopwsrv()` can not be directly back-enabled by STREAMS because there is no direct `queue_t` linkage between the two Streams. Note that no message ever gets queued to the read service procedure. Messages are kept on the write side so that flow control can propagate up to the Stream head. There is a defensive check to see if the cross-connect has broken. `qenable()` schedules the write side of the other Stream.

Close

`loopclose()` breaks the connection between the Streams.

```
static int loopclose(q)
queue_t *q;
{
    register struct loop *loop;

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

    /*
     * If we are connected to another stream, break the
     * linkage, and send a hangup message.
     * The hangup message causes the stream head to fail writes,
     * allow the queued data to be read completely, and then
     * return EOF on subsequent reads.
     */

    if (loop->oqptr) {
        ((struct loop *)loop->oqptr->q_ptr)->qptr = NULL;
```

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

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

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

11.8. Multiplexing

Multiplexing Configurations

This section describes how STREAMS multiplexing configurations are created and discusses multiplexing drivers. A STREAMS multiplexor is a pseudo-driver with multiple Streams connected to it. The primary function of the driver is to switch messages among the connected Streams. Multiplexor configurations are created from user level by system calls. The *Other Facilities*, section of the *Introduction to STREAMS* contains the required introduction to STREAMS multiplexing.

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

Multiple Streams can be connected above a driver by use of `open(2)` calls. This was done for the loop-around driver of the previous section and for the driver handling multiple minor devices in the *Drivers* section. There is no difference between the connections to these drivers, only the functions performed by the driver are different. In the multiplexing case, the driver routes data between multiple Streams. In the device driver case, the driver routes data between user processes and associated physical ports. Multiplexing with Streams connected above will be referred to as an N-to-1, or upper, multiplexor. STREAMS does not provide any facilities beyond `open()` and `close(2)` to connect or disconnect upper Streams for multiplexing purposes.

From the driver's perspective, upper and lower configurations differ only in the way they are initially connected to the driver. The implementation requirements are the same: route the data and handle flow control. All multiplexor drivers require special developer-provided software to perform the multiplexing data routing and to handle flow control. STREAMS does not directly support flow control among multiple Streams.

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

As discussed in the *Drivers* section, the multiple Streams that represent minor devices are actually distinct Streams in which the driver keeps track of each Stream attached to it. The Streams are not really connected to their common driver. The same is true for STREAMS multiplexors of any configuration. The multiplexed Streams are distinct and the driver must be implemented to do most of the work. As stated above, the only difference between configurations is the manner of connecting and disconnecting. Only lower connections have use of the multiplexing facility.

Connecting Lower Streams

A lower multiplexor is connected as follows: The initial `open()` to a multiplexing driver creates a Stream, as in any other driver. As usual, `open()` uses the first two `streamtab` structure entries (see *Opening a Stream* in the *Streams Mechanism* section) to create the driver QUEUEs. At this point, the only distinguishing characteristic of this Stream are non-NULL entries in the `streamtab` `st_mux[rw]init` (mux) fields:

```
struct streamtab {
    struct qinit    *st_rdinit;     /* defines read QUEUE */
    struct qinit    *st_wrinit;     /* defines write QUEUE */
    struct qinit    *st_muxrinit;   /* for multiplexing drivers only */
    struct qinit    *st_muxwinit;   /* for multiplexing drivers only */
    char            **st_modlist;   /* list of modules to be pushed */
};
```

These fields are ignored by the `open()` (see the rightmost Stream in figure *Internet Multiplexor Before Connecting*). Any other Stream subsequently opened to this driver will have the same `streamtab` and thereby the same mux fields.

Next, another file is opened to create a (soon to be) lower Stream. The driver for the lower Stream is typically a device driver (see the leftmost Stream in figure *Internet Multiplexor Before Connecting*). This Stream has no distinguishing characteristics. It can include any driver compatible with the multiplexor. Any modules required on the lower Stream must be pushed onto it now.

Next, this lower Stream is connected below the multiplexing driver with an I_LINK `ioctl()` call (see the `streamio(4)` man page). As shown in figure *11-1*, all Stream components are constructed in a similar manner. The Stream head points to the stream-head-routines as its procedures (known via its `queue_t`). An I_LINK to the upper Stream, referencing the lower Stream, causes STREAMS to modify the contents of the Stream head in the lower Stream. The pointers to the stream-head-routines, and other values, in the Stream head are replaced with those contained in the mux fields of the multiplexing driver's `streamtab`. Changing the stream-head-routines on the lower Stream means that all subsequent messages sent upstream by the lower Stream's driver will, ultimately, be passed to the put procedure designated in `st_muxrinit`, the multiplexing driver. The I_LINK also establishes this upper Stream as the control Stream for this lower Stream. STREAMS remembers the relationship between these two Streams until the upper Stream is closed, or the lower Stream is unlinked.

Finally, the Stream head sends to the multiplexing driver an M_IOCTL message with `ioc_cmd` set to I_LINK (see discussions of the `iocblk` structure in the *Drivers* section, above, and in the *Kernel Structures* section of *Supplementary STREAMS Material* chapter). The M_DATA part of the M_IOCTL contains a `linkblk` structure:

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

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

An I_LINK is required for each lower Stream connected to the driver. Additional upper Streams can be connected to the multiplexing driver by `open()` calls. Any message type can be sent from a lower Stream to user process(es) along any of the upper Streams. The upper Stream(s) provides the only interface between the user process(es) and the multiplexor.

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

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

Finally, note that the absence of direct linkage between the upper and lower Streams means that STREAMS flow control has to be handled by special code in the multiplexing driver. The flow control mechanism cannot see across the driver.

In general, multiplexing drivers should be implemented so that new Streams can be dynamically connected to, and existing Streams disconnected from, the driver without interfering with its ongoing operation. The number of Streams that can be connected to a multiplexor is developer dependent. However, there is a system limit, NMUXLINK, to the number of Streams that can be linked in the system.

Disconnecting Lower Streams

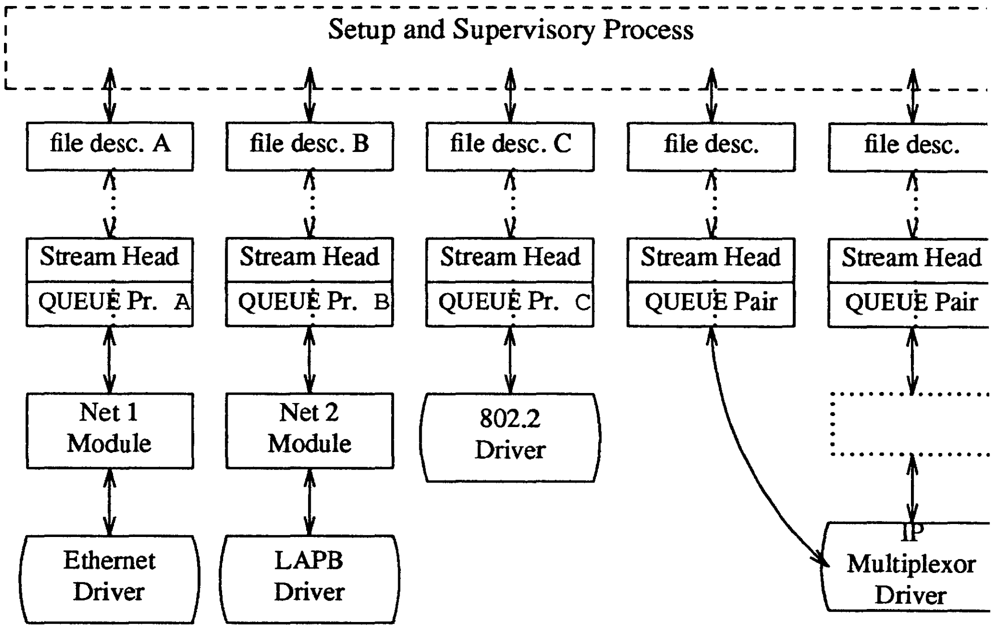
Dismantling a lower multiplexor is accomplished by disconnecting (unlinking) the lower Streams. Unlinking can be initiated in three ways: an I_UNLINK `ioctl()` referencing a specific Stream, an I_UNLINK indicating all lower Streams, or the last `close()` (i.e., causes the associated file to be closed) of the control Stream. As in the link, an unlink sends a `linkblk` structure to the driver in an M_IOCTL message. The I_UNLINK call, which unlinks a single Stream, uses the `l_index` value returned in the I_LINK to specify the lower Stream to be unlinked. The latter two calls must designate a file corresponding to a control Stream which causes all the lower Streams that were previously linked by this control Stream to be unlinked. However, the driver sees a series of individual unlinks.

If the file descriptor for a lower Stream was previously closed, a subsequent unlink will automatically close the Stream. Otherwise, the lower Stream must be closed by `close()` following the unlink. STREAMS will automatically dismantle all cascaded multiplexors (below other multiplexing Streams) if their controlling Stream is closed. An I_UNLINK will leave lower, cascaded multiplexing Streams intact unless the Stream file descriptor was previously closed.

Multiplexor Construction Example

This section describes an example of multiplexor construction and usage. A multiplexing configuration similar to the Internet figure in the *Other Facilities* section of the *Introduction to STREAMS* is discussed. Figure *Internet Multiplexor Before Connecting* shows the Streams before their connection to create the multiplexing configuration of figure *Internet Multiplexor After Connecting*. Multiple upper and lower Streams interface to the multiplexor driver. The user processes of figure *Internet Multiplexor After Connecting* are not shown in figure *Internet Multiplexor Before Connecting*.

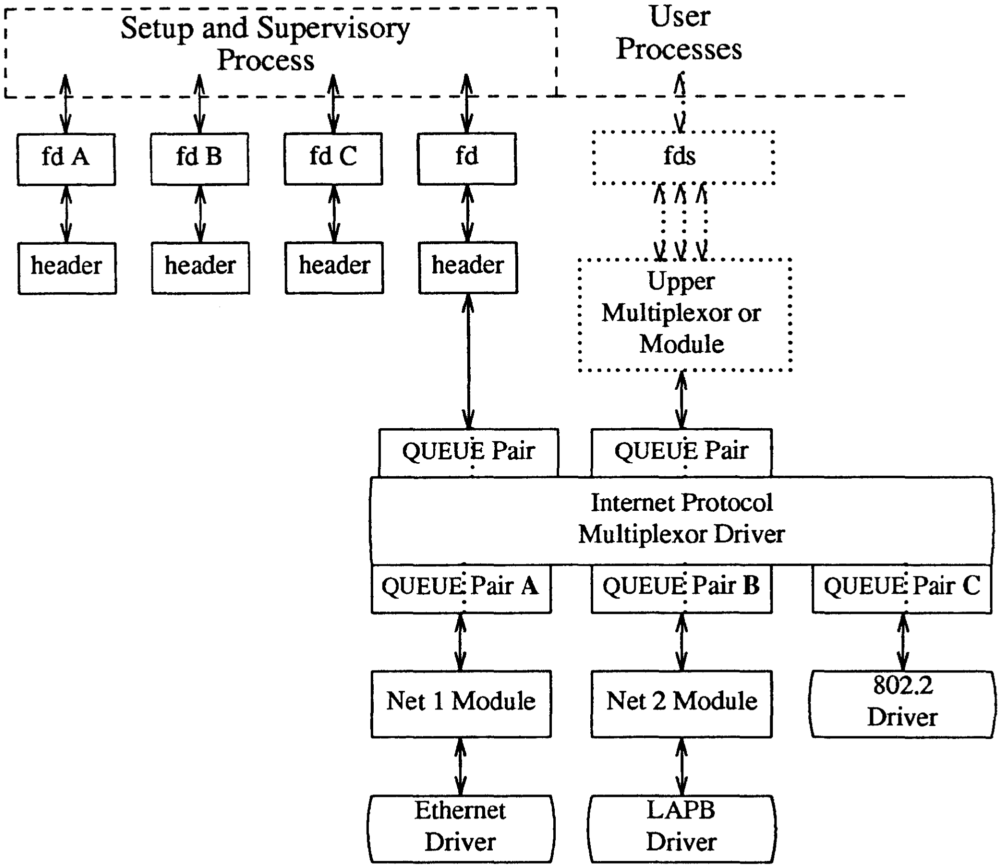
Figure 11-7 *Internet Multiplexor Before Connecting*


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

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

Figure *Internet Multiplexor Before Connecting* depicts only a portion of the full, larger Stream. As shown in the dotted rectangle above the IP multiplexor, there generally would be an upper TCP multiplexor, additional modules and, possibly, additional multiplexors in the Stream. Multiplexors could also be cascaded below the IP driver if the device drivers were replaced by multiplexor drivers.

Figure 11-8 *Internet Multiplexor After Connecting*



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

Streams A, B and C are opened by the process, and modules are pushed as needed. Two upper Streams are opened to the IP multiplexor. The rightmost Stream represents multiple Streams, each connected to a process using the network. The Stream second from the right provides a direct path to the multiplexor for supervisory functions. It is the control Stream, leading to a process which sets up and supervises this configuration. It is always directly connected to the IP driver. Although not shown, modules can be pushed on the control Stream.

After the Streams are opened, the supervisory process typically transfers routing information to the IP drivers (and any other multiplexors above the IP), and initializes the links. As each link becomes operational, its Stream is connected below the IP driver. If a more complex multiplexing configuration is required, the IP multiplexor Stream with all its connected links can be connected below another multiplexor driver.

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

Multiplexing Driver

This section contains an example of a multiplexing driver that implements an N-to-1 configuration. This configuration might be used for terminal windows, where each transmission to or from the terminal identifies the window. This resembles a typical device driver, with two differences: the device handling functions are performed by a separate driver, connected as a lower Stream, and the device information (i.e., relevant user process) is contained in the input data rather than in an interrupt call.

Each upper Stream is connected by an `open(2)`. A single lower Stream is opened and then it is linked by use of the multiplexing facility. This lower Stream might connect to the tty driver. The implementation of this example is a foundation for an M to N multiplexor.

As in the loop-around driver, flow control requires the use of standard and special code, since physical connectivity among the Streams is broken at the driver. Different approaches are used for flow control on the lower Stream, for messages coming upstream from the device driver, and on the upper Streams, for messages coming downstream from the user processes.

The multiplexor declarations are:

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

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

static struct module_info info = {
    0, "mux", 0, INFPSZ, 512, 128
};
static struct qinit urinit = {    /* upper read */
    NULL, NULL, muxopen, muxclose, NULL, &info, NULL
};
static struct qinit uwinit = {    /* upper write */
    muxuwput, NULL, NULL, NULL, NULL, &info, NULL
};
static struct qinit lrinit = {    /* lower read */
    muxlrput, NULL, NULL, NULL, NULL, &info, NULL
};
static struct qinit lwinit = {    /* lower write */
    NULL, muxlwsrv, NULL, NULL, NULL, &info, NULL
};

struct streamtab muxinfo =
    {&urinit, &uwinit, &lrinit, &lwinit};
```

```
struct mux {
    queue_t *qptr;    /* back pointer to read queue  */
};

extern struct mux mux_mux[ ];
extern int mux_cnt;

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

static queue_t *get_next_q();
```

The four `streamtab` entries correspond to the upper read, upper write, lower read, and lower write `qinit` structures. The multiplexing `qinit` structures replace those in each (in this case there is only one) lower Stream head after the I_LINK has completed successfully. In a multiplexing configuration, the processing performed by the multiplexing driver can be partitioned between the upper and lower QUEUEs. There must be an upper Stream write, and lower Stream read, put procedures. In general, only upper write side and lower read side procedures are used. Application specific flow control requirements might modify this. If the QUEUE procedures of the opposite upper/lower QUEUE are not needed, the QUEUE can be skipped over, and the message put to the following QUEUE.

In the example, the upper read side procedures are not used. The lower Stream read QUEUE put procedure transfers the message directly to the read QUEUE upstream from the multiplexor. There is no lower write put procedure because the upper write put procedure directly feeds the lower write service procedure, as described below.

The driver uses a private data structure, `mux`. `mux_mux[dev]` points back to the opened upper read QUEUE. This is used to route messages coming upstream from the driver to the appropriate upper QUEUE. It is also used to find a free minor device for a CLONEOPEN driver open case.

The upper QUEUE open contains the canonical driver open code:

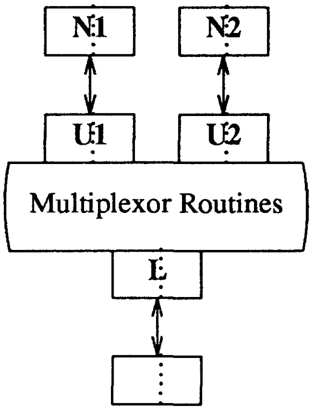
```
static int muxopen(q, dev, flag, sflag)
queue_t *q;
{
    struct mux *mux;

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

    if (dev >= mux_cnt)
        return OPENFAIL;

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

`muxopen` checks for a clone or ordinary open call. It loads `q_ptr` to point at the `mux_mux[]` structure.

The core multiplexor processing is the following: downstream data written to an upper Stream is queued on the corresponding upper write message queue. This allows flow control to propagate towards the Stream head for each upper Stream. However, there is no service procedure on the upper write side. All M_DATA messages from all the upper message queues are ultimately dequeued by the service procedure on the lower (linked) write side. The upper write Streams are serviced in a round-robin fashion by the lower write service procedure. A lower write service procedure, rather than a write put procedure, is used so that flow control, coming up from the driver below, may be handled.

On the lower read side, data coming up the lower Stream is passed to the lower read put procedure. The procedure routes the data to an upper Stream based on the first byte of the message. This byte holds the minor device number of an upper Stream. The put procedure handles flow control by testing the upper Stream at the first upper read QUEUE beyond the driver. That is, the put procedure treats the Stream component above the driver as the next QUEUE.

Figure 11-9 *Example Multiplexor Configuration*



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

Upper Write Put Procedure

`muxuwput`, the upper QUEUE write put procedure, traps ioctls, in particular I_LINK and I_UNLINK:

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

{

    int s;
    struct mux *mux;

    mux = (struct mux *)q->q_ptr;
    switch (mp->b_datap->db_type) {
    case M_IOCTL: {
        struct iocblk *iocp;
        struct linkblk *linkp;

        /*
         * Ioctl. Only channel 0 can do ioctls. Two
         * calls are recognized: LINK, and UNLINK
         */

        if (mux != mux_mux)
            goto iocnak;

        iocp = (struct iocblk *) mp->b_rptr;
        switch (iocp->ioc_cmd) {
        case I_LINK:

            /*
             * Link. The data contains a linkblk structure
             * Remember the bottom queue in muxbot.
             */

            if (muxbot != NULL)
                goto iocnak;
            linkp = (struct linkblk *) mp->b_cont->b_rptr;
            muxbot = linkp->l_qbot;
            muxerr = 0;
            mp->b_datap->db_type = M_IOCACK;
            iocp->ioc_count = 0;
            qreply(q, mp);
            break;
        case I_UNLINK:
            /*
             * Unlink. The data contains a linkblk structure.
             * Should not fail an unlink. Null out muxbot.
             */

            linkp = (struct linkblk *) mp->b_cont->b_rptr;
            muxbot = NULL;
            mp->b_datap->db_type = M_IOCACK;
            iocp->ioc_count = 0;
            qreply(q, mp);
            break;
        default:
        iocnak:

            /* fail ioctl */

            mp->b_datap->db_type = M_IOCNAK;
            qreply(q, mp);
        }

        break;

    }
```

First, there is a check to enforce that the Stream associated with minor device 0 will be the single, controlling Stream. Ioctls are only accepted on this Stream. As described previously, a controlling Stream is the one that issues the I_LINK. Having a single control Stream is a recommended practice. I_LINK and I_UNLINK include a `linkblk` structure, described previously, containing:

`l_qtop`

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

`l_qbot`
The new lower write QUEUE. It is the former Stream head write QUEUE. It is of most interest since that is where the multiplexor gets and puts its data.

`l_index`
A unique (system wide) identifier for the link. It can be used for routing, or during selective unlinks, as described above. Since the example only supports a single link, `l_index` is not used.

For I_LINK, `l_qbot` is saved in `muxbot` and an ack is generated. From this point on, until an I_UNLINK occurs, data from upper queues will be routed through `muxbot`. Note that when an I_LINK, is received, the lower Stream has already been connected. This allows the driver to send messages downstream to perform any initialization functions. Returning an M_IOCNAK message (nak) in response to an I_LINK will cause the lower Stream to be disconnected.

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

`muxuwput` handles M_FLUSH messages as a normal driver would:

```
case M_FLUSH:
     if (*mp->b_rptr & FLUSHW)
          flushq(q, FLUSHDATA);
     if (*mp->b_rptr & FLUSHR) {
          flushq(RD(q), FLUSHDATA);
          *mp->b_rptr &= ~FLUSHW;
          qreply(q, mp);
     } else
          freemsg(mp);
     break;
case M_DATA:
     /*
      * Data. If we have no bottom queue --> fail
      * Otherwise, queue the data, and invoke the lower
      * service procedure.
      */
     if (muxerr || muxbot == NULL)
          goto bad;
     putq(q, mp);     /* place message on upper write message queue */
     qenable(muxbot);     /* lower service write procedure */
     break;
default:
bad:
     /*
      * Send an error message upstream.
      */
     mp->b_datap->db_type = M_ERROR;
     mp->b_rptr = mp->b_wptr = mp->b_datap->db_base;
     *mp->b_wptr++ = EINVAL;
     qreply(q, mp);
```

```
    }
}
```

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

Lower **QUEUE** Write Service Procedure

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

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

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

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

Finding messages on upper write queues is handled by `get_next_q()`:

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

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

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

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

    return NULL;
}
```

`get_next_queue()` searches the upper queues in a round robin fashion looking for the first one containing a message. It returns the `queue_t` pointer or

NULL if there is no work to do.

Lower Read Put Procedure

The lower (linked) queue read put procedure is:

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

    switch(mp->b_datap->db_type) {
    case M_FLUSH:

        /*
         * Flush queues. NOTE: sense of tests is reversed
         * since we are acting like a "stream head"
         */

        if (*mp->b_rptr & FLUSHR)
            flushq(q, 0);
        if (*mp->b_rptr & FLUSHW) {
            *mp->b_rptr &= ~FLUSHR;
            qreply(q, mp);
        } else
            freemsg(mp);
        break;

    case M_ERROR:
    case M_HANGUP:
        muxerr = 1;
        freemsg(mp);
        break;

    case M_DATA:
        /*
         * Route message. First byte indicates
         * device to send to. No flow control.
         *
         * Extract and delete device number. If the leading block is
         * now empty and more blocks follow, strip the leading block.
         * The stream head interprets a leading zero length block
         * as an EOF regardless of what follows (sigh).
         */

        dev = *mp->b_rptr++;
        if (mp->b_rptr == mp->b_wptr &&
          (b_cont = mp->b_cont)) {
            freeb(mp);
            mp = b_cont;
        }
        /* Sanity check. Device must be in range */

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

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

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

`muxlrput` receives messages from the linked Stream. In this case, it is acting as a Stream head. It handles M_FLUSH messages. Note the code is reversed from that of a driver, handling M_FLUSH messages from upstream.

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

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

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

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

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

11.9. Service Interface

Definition

STREAMS provides the means to implement a service interface between any two components in a Stream, and between a user process and the topmost module in the Stream. A service interface is defined at the boundary between a service user and a service provider. A service interface is a set of primitives and the rules for the allowable sequences of primitives across the boundary. These rules are typically represented by a state machine. In STREAMS, the service user and provider are implemented in a module, driver, or user process. The primitives are carried bidirectionally between a service user and provider in M_PROTO and M_PCPROTO (generically, PROTO) messages. M_PCPROTO is the priority version of M_PROTO.

Message Usage

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

PROTO messages can be sent bidirectionally (up and downstream) on a Stream and bidirectionally between a Stream and a user process. `putmsg(2)` and `getmsg(2)` system calls are analogous, respectively, to `write(2)` and `read(2)` except that the former allow both data and control parts to be (separately) passed, and they observe message boundary alignment across the user-Stream boundary. `putmsg()` and `getmsg()` separately copy the control part (M_PROTO or M_PCPROTO block) and data part (M_DATA blocks) between the Stream and user process.

An M_PCPROTO message is normally used to acknowledge M_PROTO messages and not to carry protocol expedited data. M_PCPROTO insures that the acknowledgement reaches the service user before any other message. If the service user is a user process, the Stream head will only store a single M_PCPROTO message, and discard subsequent M_PCPROTO messages until the first one is read with `getmsg(2)`.

The following rules pertain to service interfaces:

- Modules and drivers that support a service interface must act upon all PROTO messages and not pass them through.
- Modules may be inserted between a service user and a service provider to manipulate the data part as it passes between them. However, these modules may not alter the contents of the control part (PROTO block, first message block) nor alter the boundaries of the control or data parts. That is, the message blocks comprising the data part may be changed, but the message may not be split into separate messages nor combined with other messages. In addition, modules and drivers must observe the rule that priority messages are not subject to flow control and forward them accordingly (e.g., see the beginning of `modwsrv()` in the *Message Queues and Service Procedures* section). Priority messages also bypass flow control at the user-Stream boundary [e.g., see `putmsg(2)`].

Example

The example below is part of a module which illustrates the concept of a service interface. The module implements a simple datagram interface.

Declarations

The service interface primitives are defined in the declarations:

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

/*
 * Primitives initiated by the service user:
 */
#define BIND_REQ         1    /* bind request   */
#define UNITDATA_REQ     2    /* unitdata request    */
/*
 * Primitives initiated by the service provider:
 */
#define OK_ACK           3    /* bind acknowledgment    */
#define ERROR_ACK        4    /* error acknowledgment    */
#define UNITDATA_IND     5    /* unitdata indication    */
/*
 * The following structures define the format of the
 * stream message block of the above primitives.
 */
struct bind_req {            /* bind request  */
    long    PRIM_type;       /* always BIND_REQ  */
    long    BIND_addr;       /* addr to bind     */
};
```

```
struct unitdata_req {     /* unitdata request */
    long    PRIM_type;    /* always UNITDATA_REQ */
    long    DEST_addr;    /* dest addr        */
};
struct ok_ack {           /* ok acknowledgment */
    long    PRIM_type;    /* always OK_ACK */
};
struct error_ack {        /* error acknowledgment */
    long    PRIM_type;    /* always ERROR_ACK */
    long    UNIX_error;   /* SunOS error code  */
};
struct unitdata_ind {     /* unitdata indication */
    long    PRIM_type;    /* always UNITDATA_IND */
    long    SRC_addr;     /* source addr      */
};
union primitives {        /* union of all primitives */
    long  type;
    struct bind_req bind_req;
    struct unitdata_req unitdata_req;
    struct ok_ack ok_ack;
    struct error_ack error_ack;
    struct unitdata_ind unitdata_ind;
};
struct dgproto {          /* structure per minor device */
    short state;          /* current provider state */
    long addr;        /* net address */
};
/* Provider states */

#define IDLE    0
#define BOUND   1
```

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

Two commands are recognized by the module:

BIND_REQ

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

UNITDATA_REQ

Send a datagram to the specified address.

Three messages are generated:

OK_ACK

A positive acknowledgement (ack) of BIND_REQ.

ERROR_ACK

A negative acknowledgement of BIND_REQ.

UNITDATA_IND
A datagram from the network has been received (this code is not shown).

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

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

state
current state of the Stream (endpoint) IDLE or BOUND

addr
network address that has been bound to this Stream

It is assumed (though not shown) that the module open procedure sets the write queue `q_ptr` to point at one of these structures.

Service Interface Procedure

The write put procedure is:

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

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

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

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

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

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

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

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

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

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

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

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

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

For UNITDATA_REQ, the state is also checked; it must be BOUND. As above, the message size and destination address are checked. If there is any error, the message is simply discarded. (This action may seem rash, but it is in accordance with the interface specification, which is not shown. Another specification might call for the generation of a UNITDATA_ERROR indication.) If all is well, the data part of the message, if it exists, is put on the queue, and the lower half of the driver is started.

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

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

11.10. Advanced Topics

Recovering From No Buffers

The `bufcall()` utility (see *Utilities* in the *Supplementary STREAMS Material* chapter) is used to recover from an `allocb()` failure. The call syntax is as follows:

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

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

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

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

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

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

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

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

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

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

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

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

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

    while (mp = getq(q)) {

        /* check for priority messages and canput ... */

        /*
         * Allocate a header to prepend to the message. If
         * the allocb fails, use bufcall to reschedule ourself.
         */
        if ((bp = allocb(HDRSZ, BPRI_MED)) == NULL) {
            if (!bufcall(HDRSZ, BPRI_MED, qenable, q)) {
                /*
                 * The bufcall request has failed. Discard
                 * the message and keep running to avoid hanging.
                 */
                freemsg(mp);
                continue;
            }
            /*
             * Put the message back and exit, we will be re-enabled later
             */
            putbq(q, mp);
            return;
        }

        /* process message .... */

    }
}
```

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

Advanced Flow Control

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

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

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

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

Signals

STREAMS allows modules and drivers to cause a signal to be sent to user process(es) through an M_SIG or M_PCSIG message (see *Message Types* in the *Supplementary STREAMS Material* chapter) sent upstream. M_PCSIG is a priority version of M_SIG. For both messages, the first byte of the message specifies the signal for the Stream head to generate. If the signal is not SIGPOLL [see `signal(2)` and `sigset(2)`], then the signal is sent to the process group associated with the Stream (see below). If the signal is SIGPOLL, the signal is only sent to processes that have registered for the signal by using the I_SETSIG `ioctl(2)` (see also the `streamio(4)` call).

A process group is associated with a stream during the open of the driver or module. If the `NEWCTTY` flag is ORed into the value returned by the `open()` procedure, the process on whose behalf the module or driver is being opened has become a "session process group leader" by executing the `setspgldr()` call (which is executed by the `setpgrp()` call in the System V environment, but not in the 4BSD environment). If that process does not already have a controlling tty, and the stream does not already have a process group, then the stream is assigned to the process group that the process is the leader of and becomes that process' controlling tty.

If the driver or module wants to have a process group associated with the stream, it should OR the `NEWCTTY` flag into its return value.

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

Control of Stream Head Processing

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

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

Read Options

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

byte-stream (**RNORM**)

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

message non-discard (**RMSGN**)

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

message discard (**RMSGD**)

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

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

Write Offset

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

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

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

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

12

SunOS STREAMS Topics

SunOS STREAMS Topics **307**

12.1. Configuring STREAMS Drivers 307
  - Module Configuration 308
  - Tunable Parameters 309
  - System Error Messages 310

12.2. STREAMS in SunOS 311
  - STREAM Modules 311
  - SunOS STREAMS Extension 312
  - STREAMS Portability 312
  - User Line Disciplines 312

12

SunOS STREAMS Topics

12.1. Configuring STREAMS Drivers

The configuration of STREAMS device drivers is not fundamentally different from the configuration of regular device drivers. This section, therefore, presumes familiarity with the *Configuring the Kernel* section of this manual, which explains in some detail how new drivers are configured into the kernel.

Note that, while STREAMS give programmers a good deal of flexibility in regard to configuration issues, STREAMS drivers and protocol modules must still be precomplied into the kernel. STREAMS drivers are *not* dynamically loadable.

SunOS STREAMS drivers use exactly the same *autoconfiguration* interface as do regular SunOS drivers. This interface is designed to allow drivers (and modules) to easily define their per-instance data structures, using the information supplied by `config`. However, if a given driver or module chooses to use some other scheme for allocating its resources (such as using `kmem_alloc()` when a previously unopened device is opened), it is free to do so. This differs significantly from the System V driver/kernel interface, which arranges for such storage to be allocated elsewhere.

Each character device that is configured into the Sun kernel results in an entry being placed in the kernel `cdevsw` table. Entries for STREAMS drivers are no exception — they too are placed in `cdevsw`. However, since system calls to STREAMS drivers must be processed by the STREAMS routines, their `cdevsw` interface differs from that of non-STREAMS drivers. `config`, it should be noted, knows nothing about STREAMS drivers. It handles them correctly because, as far it it's concerned, they are just regular character drivers. There is nothing in the format of entries in a config file that distinguishes STREAMS devices/modules from other character devices.

There is, however, a difference between STREAMS and non-STREAMS `cdevsw` entries, in that STREAMS entries have only the `d_str` field set while other entries *never* have this field set. `d_str` provides the appropriate single entry point for all system calls on STREAMS files, as shown below:

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

The `d_str` entry name is formed by appending the string "info" to the STREAMS driver prefix. The "info" entry is a pointer to the driver/module declared `streamtab` structure (see *Kernel Structures*). The `streamtab` structure contains pointers to the `qinit` structures for the driver/module's read and write queues. Its declaration must be externally visible:

```
struct streamtab xxinfo = { ...
```

If the driver declares a `streamtab` named *xx*info, the `d_str` entry will contain a non-NULL pointer and the kernel will recognize the driver as a STREAMS driver and will call it by way of the appropriate STREAMS routines. If the `d_str` entry is `NULL`, the normal character I/O `cdevsw` interface will be used. Note that only `streamtab` must be externally visible in STREAMS drivers and modules, since it is used to uniquely identify the appropriate open, close, put, service and administration routines. These driver/module routines should generally be declared `static`.

Module Configuration

When adding a new STREAMS module to a kernel, one must add an entry to the `fmodsw` array in `/sys/sun/str_conf.c`. This file is analogous to `/sys/sun/conf.c` (see the *Configuring the Kernel* chapter) and its entries should be similarly conditional on the number of module instances being positive. For example, for the *xx* device:

```
#if NXX > 0
extern struct streamtab xx_info;
#endif
    .
    .
    .
struct fmodsw fmodsw[] =
{
    .
    .
    .
#if NXX > 0
    { "xx", &xx_info),
#endif
    .
    .
    .
}
```

The first of the two fields in each `fmodsw` entry is the name of the module, which will be used in all STREAMS-related `ioctl()` calls upon this module. The second is a pointer to the module's `streamtab` structure.

Tunable Parameters

Certain system parameters referenced by STREAMS are configurable when building a new kernel. They can be reset from their default values, values which are calculated to correspond to the value of `MAXUSERS`, by using the config file `OPTIONS` mechanism. (See `config`(8)). In this discussion, the term "queues" refers to `queue_t` structures. The tunable parameters are:

NSTREAM
Total number of Streams that may be open at one time in a system.

NBLK4096
Total number of 4096 byte data blocks available for STREAMS operations. The pool of data blocks is a system-wide resource, so enough blocks must be configured to satisfy all Streams.

NBLK2048
Total number of 2048 byte data blocks available for STREAMS operations.

NBLK1024
Total number of 1024 byte data blocks available for STREAMS operations.

NBLK512
Total number of 512 byte data blocks available for STREAMS operations.

NBLK256
Total number of 256 byte data blocks available for STREAMS operations.

NBLK128
Total number of 128 byte data blocks available for STREAMS operations.

NBLK64
Total number of 64 byte data blocks available for STREAMS operations.

NBLK16
Total number of 16 byte data blocks available for STREAMS operations.

NBLK4
Total number of 4 byte data blocks available for STREAMS operations.

NMUXLINK
Total number of Streams in system that can be linked as lower Streams to multiplexor drivers (by an `I_LINK ioctl`(2), see `streamio`(4)).

NSTREVENT
Initial number of internal event cells available in system to support `bufcall()` and `poll(2)` calls.

MAXSEPGCNT
The number of additional pages of memory that can be dynamically allocated for event cells. If this value is 0, only the allocation defined by `NSTREVENT` is available for use. If the value is not 0 and if the kernel runs out of event cells, it will under some circumstances attempt to allocate an

extra page of memory from which new event cells can be created. `MAXSEPGCNT` places a limit on the number of pages that can be allocated for this purpose. Once a page has been allocated for event cells, however, it cannot be recovered later for use elsewhere.

NSTRPUSH
Maximum number of modules that may be pushed onto a single Stream.

STRMSGSZ
Maximum bytes of information that a single system call can pass to a Stream to be placed into the data part of a message (in `M_DATA` blocks). Any `write(2)` exceeding this size will be broken into multiple messages. A `putmsg(2)` with a data part exceeding this size will fail.

STRCTLSZ
Maximum bytes of information that a single system call can pass to a Stream to be placed into the control part of a message (in an `M_PROTO` or `M_PCPROTO` block). A `putmsg(2)` with a control part exceeding this size will fail.

STRLOFRAC
The percentage of data blocks of a given class at which low priority block allocation requests are automatically failed. For example, if `STRLOFRAC` is 80 and there are 48 256-byte blocks, a low priority allocation request will fail when more than 38 256-byte blocks are already allocated. This value is used to prevent deadlock situations in which a low priority activity might starve out more important functions. For example, if `STRLOFRAC` is 80 and there are 100 blocks of 256 bytes, then when more than 80 of such blocks are allocated, any low priority allocation request will fail. This value must be in the range `0~<=~STRLOFRAC~<=~STRMEDFRAC`.

STRMEDFRAC
The percentage of data blocks of a given class at which medium priority block allocation requests are automatically failed.

System Error Messages

Messages are reported to the console as a result of various error conditions detected by STREAMS. These messages and the action to be taken on their occurrence are described below. In certain cases, a tunable parameter (see previous section) may have to be changed.

stropen: out of streams
A Stream head data structure could not be allocated during the `open()` of a STREAMS device. If this occurs repeatedly, increase `NSTREAM`.

allocq: out of queues
A pair of queues could not be allocated for the Stream head during the `open()` of a driver, or a pair of queues could not be allocated for a pushable module (`I_PUSH ioctl`). This error message should never be seen, as additional space for queues is allocated dynamically when needed.

strinit: can not allocate stream data blocks
During system initialization, the system was unable to allocate enough memory for the STREAMS data blocks. The system must be rebuilt with

fewer data blocks specified.

bufcall: could not allocate stream event
A call to `bufcall()` has failed because all Stream event cells have been allocated. If this occurs repeatedly, increase `NSTREVENT`.

munlink: could not perform ioctl, closing anyway
A linked multiplexor could not be unlinked when the controlling Stream for that link was closed. The linked Stream will be unlinked and the controlling Stream will be closed anyway.

12.2. STREAMS in SunOS

SunOS 4.0 includes reimplementations of two fundamental system mechanisms in terms of STREAMS. These are:

1. The system terminal driver, which controls serial-line I/O, and
2. The Network Interface Tap (NIT) mechanism, which permits a process to talk to the "raw" Ethernet. NIT is the only networking facility which is thus far implemented in terms of STREAMS, though a TCP/IP implementation that can be accessed via STREAMS is planned.

STREAM Modules

The following STREAMS modules, necessary to support the tty driver and the Network Interface Tap, are included in SunOS 4.0.

- The ''standard tty driver'' module, which implements most of the standard tty driver behavior; it's a replacement for the current standard tty line discipline. (See `tty_std`(4M)).
- The ''ioctl mapping'' module, which maps old V7 and 4BSD `ioctl()` calls into new-style `ioctl()` calls. This gets pushed on top of the standard tty driver module, giving a stream that responds either to the old-style or new-style `ioctl()` calls. (See `tty_compat`(4M)).
- The keyboard and mouse modules, which replace the old keyboard and mouse line disciplines. (See `kb(4M)` and `ms(4M)`.
- The NIT ''packet filter'' module, which is given a set of criteria for selecting Ethernet packets, and passes only the selected packets upstream, discarding the others. Thus, the Reverse ARP daemon could request that it receive only Reverse ARP packets; filtering can be done more efficiently in this fashion than if all packets were handed to the program and it had to do the filtering itself. This also makes it easier to handle a high rate of arrival of packets, since the program doesn't have to handle the ones it's not interested in. (See `nit_pf`(4M)).
- The NIT ''buffering'' module, which buffers up received Ethernet packets and delivers them to the user program in a single chunk. Such buffering reduces the number of `read()` calls done while monitoring the Ethernet, as is necessary when the rate at which packets arrive is very high. (See `nit_buf`(4M)).

SunOS STREAMS Extension

In order to support STREAMS terminal and pseudo-terminal drivers, SunOS has extended the AT&T STREAMS mechanism. SunOS STREAMS includes a mechanism by which STREAMS drivers can specify a list of STREAMS modules to be automatically pushed onto the stream at device open time. This (or a similar) feature is necessary to allow tty drivers to present an interface compatible with that which existed in previous system releases.

STREAMS Portability

The set of internal interfaces and utility routines defined by the SunOS kernel differs considerably from that defined by the System V kernel. The STREAMS/kernel interface is well specified, however, and System V STREAMS modules and drivers that use only the interfaces it defines (see *Accessible Symbols and Functions* in the *Supplementary STREAMS Material* chapter of this manual) should be able to be adapted to the SunOS kernel without many problems. However, it's easy to use kernel facilities (data structures and routines) other than those defined in the STREAMS interface. Any such use is likely to be non-portable between System V and SunOS.

Similarly, STREAMS modules and drivers written for SunOS will only be portable to System V systems if their kernel interfaces are confined to the explicitly listed *Accessible Symbols and Functions*. If System V-compatibility is not an issue, then STREAMS modules and drivers can use any of the driver-support routines listed in the *Kernel Support Routines* appendix.

Note that STREAMS drivers, as opposed to modules, will always require a certain degree of rewriting for use on System V machines, since the SunOS autoconfiguration interface differs significantly from that used in System V. See the *The Bus-Resource Interface* section of this manual for the details of the Sun interface.

User Line Disciplines

Note that user-built line disciplines will have to be converted into STREAMS form before they will be compatible with release 4.0. This is because they probably access tty-specific internal structures, such as `clist` buffers. These structures no longer exist, having been replaced by STREAMS structures, so any routines that access them will no longer work. For information on how to proceed with the conversion of a line discipline, contact the Sun consulting department.

Character drivers that do not implement line disciples can also be converted to STREAMS form, though in this case the conversion is entirely optional. This is because the SunOS STREAMS implementation preserves the *external* interfaces to the character devices and drivers (e.g. through the standard tty compatibility module, `tty_compat(4M)`, that implements most of the 4BSD tty interfaces under STREAMS). Thus, drivers which do not directly access underlying system data structures will continue to work without changes.

Drivers that have fancy read and write routines (routines that do anything more than just import parameters and perhaps start another routine) are probably not good candidates for conversion into STREAMS form, since STREAMS read/write modules should just set up data for the STREAMS queues.

A line-printer driver is an example of a character driver that could be written in terms of STREAMS, but doesn't need to be, and doesn't need to be converted to

STREAMS if it already exists. After all, while a line-printer driver does transform a stream of characters (this transformation could certainly be built into a STREAMS module), its transformation is unlikely to be of interest to other programs. Thus, there's little to be gained by encapsulating it in a module. And, since line-printer drivers implement no line discipline, they will continue to work with SunOS 4.0.

A

Supplementary STREAMS Material

Supplementary STREAMS Material **317**

A.1. Kernel Structures 317

`streamtab` 317

`QUEUE` Structures 317

A.2. Message Structures 318

`iocblk` 319

`linkblk` 319

A.3. Message Types 320

Ordinary Messages 320

Priority Messages 325

A.4. Utilities 327

Buffer Allocation Priority 328

`adjmsg()` — Trim Bytes in a Message 329

`allocb()` — Allocate a Message Block 329

`backq()` — Get Pointer to Queue Behind a Given Queue 329

`bufcall()` — Recover from Failure of **`allocb`** 330

`canput()` — Test for Room in a Queue 330

`copyb()` — Copy a Message Block 330

`copymsg()` — Copy a Message 331

`datamsg()` — Test Whether Message is a Data Message 331

`dupb()` — Duplicate a Message Block Descriptor 331

`dupmsg()` — Duplicate a Message 331

`enableok()` — Re-allow Queue to be Scheduled 332

flushq() — Flush a Queue 332
freeb() — Free a Message Block 332
freemsg() — Free All Message Blocks in a Message 332
getq() — Get a Message from a Queue 332
insq() — Put a Message at a Specific Place in a Queue 333
linkb() — Concatenate Two Messages into One 333
msgdsize() — Get Number of Data Bytes in a Message 333
noenable() — Prevent a Queue from Being Scheduled 333
OTHERQ() — Get Pointer to the Mate Queue 334
pullupmsg() — Concatenate Bytes in a Message 334
putbq() — Return a Message to the Beginning of a Queue 334
putctl() — Put a Control Message 334
putctl1() — Put One-byte Parameter Control Message 335
putnext() — Put a Message to the Next Queue 335
putq() — Put a Message on a Queue 335
qenable() — Enable a Queue 336
qreply() — Send Reverse-Direction Message on Stream 336
qsize() — Find the Number of Messages on a Queue 336
RD() — Get Pointer to the Read Queue 336
rmvb() — Remove a Message Block from a Message 336
rmvq() — Remove a Message from a Queue 337
splstr() — Set Processor Level 337
strlog() — Submit Messages for Logging 337
testb() — Check for an Available Buffer 337
unlinkb() – Remove Message Block from Message Head 338
WR() — Get Pointer to the Write Queue 338
A.5. Design Guidelines 338
General Rules 338
System Calls 339
Data Structures 339
Header Files 340
Accessible Symbols and Functions 340
Rules for Put and Service Procedures 341
A.6. STREAMS Glossary 343

A

Supplementary STREAMS Material

A.1. Kernel Structures

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

STREAMS kernel structures are contained in `<sys/stream.h>`.

`streamtab`

As discussed in the *Streams Mechanism* section of the *STREAMS Module and Driver Programming* chapter, this structure defines a module or driver:

```
struct streamtab {
    struct qinit    *st_rdinit;     /* defines read QUEUE */
    struct qinit    *st_wrinit;     /* defines write QUEUE */
    struct qinit    *st_muxrinit;  /* for multiplexing drivers only */
    struct qinit    *st_muxwinit;  /* for multiplexing drivers only */
    char          **st_modlist;  /* list of modules to be pushed  */
};
```

`QUEUE` Structures

Two sets of QUEUE structures form a module. The structures, discussed in the *Streams Mechanism* and *Message Queues and Service Procedures* sections of the *STREAMS Module and Driver Programming* chapter, are `queue_t`, `qinit`, `module_info` and, optionally, `module_stat`:

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

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

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

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

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

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

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

A.2. Message Structures

As described in the *Messages* section of *STREAMS Module and Driver Programming*, a message is composed of a linked list of triples, consisting of two structures and a data buffer:

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

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

iocblk

As described in the *Drivers* section of the *STREAMS Module and Driver Programming* chapter and in *Message Types*, below, this is contained in an `M_IOCTL` message block:

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

linkblk

As described in the *Multiplexing* section of *STREAMS Module and Driver Programming*, this is used in lower multiplexor drivers:

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

A.3. Message Types

Eighteen STREAMS message types are defined. The message types differ in their intended purposes, their treatment at the Stream head, and in their message queueing priority (see the *Message Queues and Service Procedures* section of the *STREAMS Module and Driver Programming* chapter.

STREAMS does not prevent a module or driver from generating any message type and sending it in any direction on the Stream. However, established processing and direction rules should be observed. Stream head processing according to message type is fixed, although certain parameters can be altered.

The message types are described below, classified according to their message queueing priority. Ordinary messages are described first, with priority messages following. In certain cases, two message types may perform similar functions, differing in priority. Message construction is described in the *Messages* section of the *STREAMS Module and Driver Programming* chapter. The use of the word module will generally imply "module or driver."

Ordinary Messages

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

M_DATA

Intended to contain ordinary data. Messages allocated by the `allocb()` routine (see *Message Types*, below) are type `M_DATA` by default. `M_DATA` messages are generally sent bidirectionally on a Stream and their contents can be passed between a process and the Stream head. In the `getmsg(2)` and `putmsg(2)` system calls, the contents of `M_DATA` message blocks are referred to as the data part. Messages composed of multiple message blocks will typically have `M_DATA` as the message type for all message blocks following the first.

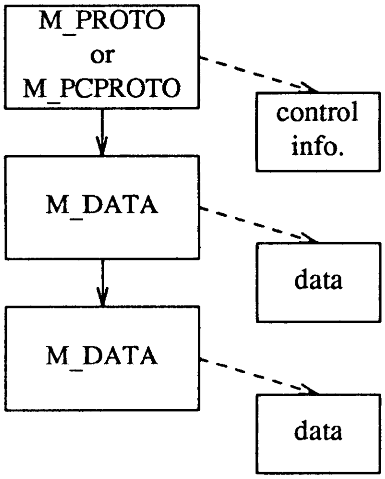
M_PROTO

Intended to contain internal control information and associated data. The message format is one `M_PROTO` message block followed by zero or more `M_DATA` message blocks as shown below: The semantics of the `M_DATA` and `M_PROTO` message block are determined by the STREAMS module that receives the message.

The `M_PROTO` message block will typically contain implementation dependent control information. `M_PROTO` messages are generally sent bidirectionally on a Stream, and their contents can be passed between a process and the Stream head. The contents of the first message block of an `M_PROTO` message is generally referred to as the control part, and the contents of any following `M_DATA` message blocks are referred to as the data part. In the `getmsg(2)` and `putmsg(2)` system calls, the control and data parts are passed separately. These calls refer to `M_PROTO` messages as non-priority messages.

Note that, although its use is not recommended, the format of `M_PROTO` and `M_PCPROTO` (generically `PROTO`) messages sent upstream to the Stream head allows multiple `PROTO` blocks at the beginning of the message. `getmsg()` will compact the blocks into a single control part when passing them to the user process.

Figure A-1 `M_PROTO` *and* `M_PCPROTO` *Message Structure*



`M_IOCTL`

Generated by the Stream head in response to an `I_STR`, and certain other, `ioctl(2)` system calls (see the `streamio(4)` man page.) When one of these `ioctl()`s is received from a user process, the Stream head uses values from the process and supplied in the call to create an `M_IOCTL` message containing them, and sends the message downstream. `M_IOCTL` messages are intended to perform the general ioctl functions of character device drivers.

The user values are supplied in a structure of the following form, provided as an argument to the `ioctl()` call (see `I_STR` in the `streamio(4)` man page.

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

where `ic_cmd` is the request (or command) defined by a downstream module or driver, `ic_timout` is the time the Stream head will wait for acknowledgement to the `M_IOCTL` message before timing out, `ic_dp` is a pointer to an optional data argument. On input, `ic_len` contains the length of the data argument passed in and, on return from the call, it contains the length of the data, if any, being returned to the user.

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

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

The `iocblk` structure is defined in `<sys/stream.h>`. `ioc_cmd` corresponds to `ic_cmd`. `ioc_uid` and `ioc_gid` are the effective user and group IDs for the user sending the `ioctl()`, and can be tested to determine if the user issuing the `ioctl()` call is authorized to do so. `ioc_count` is the number of data bytes, if any, contained in the message and corresponds to `ic_len`.

`ioc_id` is an identifier generated internally, and is used to match each `M_IOCTL` message sent downstream with a response which must be sent upstream to the Stream head. The response is contained in an `M_IOCACK` (positive acknowledgement) or an `M_IOCNAK` (negative acknowledgement) messages. Both these message types have the same format as an `M_IOCTL` message and contain an `iocblk` structure in the first block with optional data blocks following. If one of these messages reaches the Stream head with an identifier which does not match that of the currently-outstanding `M_IOCTL` message, the response message is discarded. A common means of assuring that the correct identifier is returned, is for the replying module to convert the `M_IOCTL` message type into the appropriate response type and set `ioc_count` to 0, if no data is returned. Then, the `qreply()` utility (see *Utilities*, below) is used to send the response to the Stream head.

`ioc_error` holds any return error condition set by a downstream module. If this value is non-zero, it is returned to the user in `errno`. Note that both an `M_IOCNAK` and an `M_IOCACK` may return an error. `ioc_rval` holds any `M_IOCACK` return value set by a responding module.

If a user supplies data to be sent downstream, the Stream head copies the data, pointed to by `ic_dp` in the `strioctl` structure, into `M_DATA` message blocks and links the blocks to the initial `M_IOCTL` message block. `ioc_count` is copied from `ic_len`. If there is no data, `ioc_count` is zero.

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

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

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

M_CTL

Generated by modules that wish to send information to a particular module or type of module. `M_CTL` messages are typically used for inter-module communication, as when adjacent STREAMS protocol modules negotiate the terms of their interface. An `M_CTL` message cannot be generated by a user-level process and is always discarded if passed to the Stream head.

M_BREAK

Sent to a driver to request that `BREAK` be transmitted on whatever media the driver is controlling.

The message format is not defined by STREAMS and its use is developer dependent. This message may be considered a special case of an `M_CTL` message. An `M_BREAK` message cannot be generated by a user-level process and is always discarded if passed to the Stream head.

M_DELAY

Sent to a media driver to request a real-time delay on output. The data buffer associated with this message type is expected to contain an integer to indicate the number of machine ticks of delay desired. `M_DELAY` messages are typically used to prevent transmitted data from exceeding the buffering capacity of slower terminals.

The message format is not defined by STREAMS and its use is developer dependent. Not all media drivers may understand this message. This message may be considered a special case of an `M_CTL` message. An `M_DELAY` message cannot be generated by a user-level process and is always discarded if passed to the Stream head.

M_PASSFP

This is used by STREAMS to pass a file pointer from the Stream head at one end of a Stream pipe to the Stream head at the other end of the same Stream pipe. (A Stream pipe is a Stream that is terminated at both ends by a Stream head; one end of the Stream can always find the other by following the `q_next` pointers in the Stream. The means by which such a structure is created is not described in this document.)

The message is generated as a result of an `I_SENDFD` `ioctl()` (see the `streamio(4)` man page) issued by a process to the sending Stream head. STREAMS places the `M_PASSFP` message directly on the destination Stream head's read queue to be retrieved by an `I_RECVFD` `ioctl()` (see the streamio (4) man page). The message is placed without passing it through the Stream (i.e., it is not seen by any modules or drivers in the

Stream). This message type should never be present on any queue except the read queue of a Stream head. Consequently, modules and drivers do not need to recognize this message type, and it can be ignored by module and driver developers.

M_SETOPTS

Alters some characteristics of the Stream head. It is generated by any downstream module, and is interpreted by the Stream head. The data buffer of the message has the following structure:

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

where `so_flags` specifies which options are to be altered, and can be any combination of the following:

SO_ALL

Update all options according to the values specified in the remaining fields of the `stroptions` structure.

SO_READOPT

Set the read mode (see the `read(2)` man page) to RNORM (byte stream), RMSGD (message discard), or RMSGN (message non-discard) as specified by the value of `so_readopt`.

SO_WROFF

Direct the Stream head to insert an offset specified by `so_wroff` into the first message block of all `M_DATA` messages created as a result of a `write()` system call. The same offset is inserted into the first `M_DATA` message block, if any, of all messages created by a `putmsg()` system call. The default offset is zero.

The offset must be less than the maximum message buffer size (system dependent). Under certain circumstances, a write offset may not be inserted. A module or driver must test that `b_rptr` in the `mblk_t` structure is greater than `db_base` in the `dblk_t` structure to determine that an offset has been inserted in the first message block.

SO_MINPSZ

Change the minimum packet size value associated with the Stream head read queue to `so_minpsz` (see `q_minpsz` in the `queue_t` structure, in the *Kernel Structures* section, above) This value is advisory for the module immediately below the Stream head. It is intended to limit the size of `M_DATA` messages that the module should put to the Stream head. There is no intended minimum size for other message types. The default value in the Stream head is 0.

SO_MAXPSZ

Change the maximum packet size value associated with the Stream head read queue to `so_maxpsz` (see `q_maxpsz` in the `queue_t` structure, in the *Kernel Structures* section, above). This value is advisory for the module immediately below the Stream head. It is intended to limit the size of `M_DATA` messages that the module should put to the Stream head. There is no intended maximum size for other message types. The default value in the Stream head is `INFPSZ`, the maximum STREAMS allows.

SO_HIWAT

Change the flow control high water mark on the Stream head read queue to the value specified in `so_hiwat`.

SO_LOWAT

Change the flow control low water mark (see `q_minpsz` in the `queue_t` structure, in the *Kernel Structures* section, above) on the Stream head read queue to the value specified in `so_lowat`.

M_SIG

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

If the signal is not `SIGPOLL` and the Stream containing the sending module or driver is a controlling TTY, the signal is sent to the associated process group. If the Stream does not have a process group, and the calling process does not have a controlling TTY, the Stream may become the controlling TTY for the caller's process group. This happens if the `NEWCTTY` flag is ORed into the value returned from a call to `open(2)`.

If the signal is `SIGPOLL`, it will be sent only to those processes that have explicitly registered to receive the signal (see `I_SETSIG` in the `streamio(4)` man page).

Priority Messages

Priority messages are not subject to flow control.

M_PCPROTO

This message type has the same format and characteristics as the `M_PROTO` message type, except for priority and the following additional attributes.

When an `M_PCPROTO` message is placed on a queue, its service procedure is always enabled. The Stream head will allow only one `M_PCPROTO` message to be placed in its read queue at a time. If an `M_PCPROTO` message is already in the queue when another arrives, the second message is silently discarded and its message blocks freed.

This message type is intended to allow data and control information to be sent outside the normal flow control constraints.

The `getmsg(2)` and `putmsg(2)` system calls refer to `M_PCPROTO` messages as priority messages.

M_ERROR

This message type is sent upstream by modules or drivers to report some downstream error condition. When the message reaches the Stream head, the Stream is marked so that all subsequent system calls issued to the Stream, excluding `close(2)` and `poll(2)`, will fail with `errno` set to the first data byte of the message. `POLLERR` is set if the Stream is being `poll()`ed (see the `poll(2)` man page. All processes sleeping on a system call to the Stream are awakened. An `M_FLUSH` message with an `FLUSHRW` argument is sent downstream.

M_HANGUP

This message type is sent upstream by a driver to report that it can no longer send data upstream. As example, this might be due to an error, or to a remote line connection being dropped. When the message reaches the Stream head, the Stream is marked so that all subsequent `write(2)` and `putmsg(2)` system calls issued to the Stream will fail and return an `ENXIO` error. Those `ioctl()`s that cause messages to be sent downstream are also failed. `POLLHUP` is set if the Stream is being `poll()`ed (see the `poll(2)` man page.

However, subsequent `read(2)` or `getmsg(2)` calls to the Stream will not generate an error. These calls will return any messages (according to their function) that were on, or in transit to, the Stream head read queue before the `M_HANGUP` message was received. When all such messages have been read, `read()` will return 0, and `getmsg()` will set each of its two length fields to 0.

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

M_IOCACK

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

M_IOCNAK

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

M_FLUSH

This message type requests all modules and drivers that receive it to flush their message queues (discard all messages in those queues) as indicated in the message. An `M_FLUSH` can originate at the Stream head, or in any module or driver. The first byte of the message contains flags that specify one of the following actions:

FLUSHR:
Flush the read queue of the module.

FLUSHW:
Flush the write queue of the module.

FLUSHRW:
Flush both the read and the write queue of the module.

Each module passes this message to its neighbor after flushing its appropriate queue(s), until the message reaches one of the ends of the Stream.

Drivers are expected to include the following processing for `M_FLUSH` messages. When an `M_FLUSH` message is sent downstream through the write queues in a Stream, the driver at the Stream end discards it if the message action indicates that the read queues in the Stream are not to be flushed (only `FLUSHW` set). If the message indicates that the read queues are to be flushed, the driver sets the `M_FLUSH` message flag to `FLUSHR`, and sends the message up the Stream's read queues. When a flush message is sent up a Stream's read side, the Stream head checks to see if the write side of the Stream is to be flushed. If only `FLUSHR` is set, the Stream head discards the message. However, if the write side of the Stream is to be flushed, the Stream head sets the `M_FLUSH` flag to `FLUSHW` and sends the message down the Stream's write side. *All modules that enqueue messages must identify and process this message type.*

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

M_START and **M_STOP**
These messages request devices to start or stop their output. They are intended to produce momentary pauses in a device's output, not to turn devices on or off.

The message format is not defined by STREAMS and its use is developer dependent. These messages may be considered special cases of an `M_CTL` message. These messages cannot be generated by a user-level process and each is always discarded if passed to the Stream head.

A.4. Utilities

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

The utilities contained in this appendix represent an interface that will be maintained in subsequent versions of SunOS. Other than these utilities (see also the *Accessible Symbols and Functions* section, below) functions contained in the STREAMS kernel code may change in future releases.

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

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

All structure definitions are contained in the *Kernel Structures* section, above, unless otherwise indicated. All routine references are found in this section unless otherwise indicated. The following definitions are used.

Blocked
A queue that can not be enabled due to flow control (see the *Flow Control* section in the *Introduction to STREAMS* chapter of the *System Services Overview*.

Enable
To schedule a queue.

Free
De-allocate a STREAMS storage.

Message block (bp)
A triplet consisting of an `mblk_t` structure, a `dblk_t` structure, and a data buffer. It is referenced by its `mblk_t` structure (see the *Messages* section of the *STREAMS Module and Driver Programming* chapter.

Message (mp)
One or more linked message blocks. A message is referenced by its first message block.

Message queue
Zero or more linked messages associated with a queue (`queue_t` structure).

Queue (q)
A `queue_t` structure. This is generally the same as QUEUE in the rest of this document (e.g., see the definitions for enable and schedule). When it appears with "message" in certain utility description lines, it means "message queue."

Schedule
Place a queue on the internal linked list of queues which will subsequently have their service procedure called by the STREAMS scheduler.

The word module will generally mean "module and/or driver." The phrase "next/following module" will generally refer to a module, driver, or Stream head. Message queueing priority (see the *Message Queues and Service Procedures* section of the *STREAMS Module and Driver Programming* chapters and the *Message Types* section, above) can be ordinary or Priority (to avoid "priority priority").

Buffer Allocation Priority

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

`BPRI_LO`
Low priority. At this priority, `allocb()` may fail even though the requested buffer size is available. This priority is used by the Stream head write routine to hold data associated with user calls.

BPRI_MED
Medium priority. This priority is typically used for normal data and control block allocation. As above, `allocb()` may fail at this priority even though a buffer of the requested size is available. However, for a given block size, an `BPRI_LO` `allocb()` call will fail before a `BPRI_MED` `allocb()` call.

BPRI_HI
High priority. This priority is typically used only for critical control message allocations. Calls to `allocb()` will succeed if a buffer of the appropriate size is available. Developers should exercise restraint in use of `BPRI_HI` allocation requests.

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

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

`adjmsg()` — Trim Bytes in a Message

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

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

`allocb()` — Allocate a Message Block

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

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

`backq()` — Get Pointer to Queue Behind a Given Queue

```
queue_t *backq(q)
    queue_t *q;
```

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

`bufcall()` — Recover from Failure of `allocb`

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

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

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

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

`canput()` — Test for Room in a Queue

```
int canput(q)
    queue_t *q;
```

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

`copyb()` — Copy a Message Block

```
mblk_t *copyb(bp)
    mblk_t *bp;
```

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

copymsg() — Copy a Message

```
mblk_t *copymsg(mp)
    mblk_t *mp;
```

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

datamsg() — Test Whether Message is a Data Message

```
#define datamsg(mp) ...
```

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

dupb() — Duplicate a Message Block Descriptor

```
mblk_t *dupb(bp)
    mblk_t *bp;
```

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

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

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

dupmsg() — Duplicate a Message

```
mblk_t *dupmsg(mp)
    mblk_t *mp;
```

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

`enableok()` — Re-allow Queue to be Scheduled

```
#define enableok(q) ...
```

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

`flushq()` — Flush a Queue

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

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

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

`freeb()` — Free a Message Block

```
int freeb(bp)
    mblk_t *bp;
```

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

`freemsg()` — Free All Message Blocks in a Message

```
int freemsg(mp)
    mblk_t *mp;
```

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

`getq()` — Get a Message from a Queue

```
mblk_t *getq(q)
    queue_t *q;
```

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

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

`insq()` — Put a Message at a Specific Place in a Queue

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

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

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

`linkb()` — Concatenate Two Messages into One

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

`linkb()` puts the message pointed at by *mp2* at the tail of the message pointed at by *mp1*.

`msgdsize()` — Get Number of Data Bytes in a Message

```
int msgdsize(mp)
    mblk_t *mp;
```

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

`noenable()` — Prevent a Queue from Being Scheduled

```
#define noenable(q) ...
```

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

OTHERQ() — Get Pointer to the Mate Queue

```
#define OTHERQ(q) ...
```

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

pullupmsg() — Concatenate Bytes in a Message

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

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

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

putbq() — Return a Message to the Beginning of a Queue

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

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

putctl() — Put a Control Message

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

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

`putctl1()` — Put One-byte Parameter Control Message

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

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

`putnext()` — Put a Message to the Next Queue

```
#define putnext(q, mp) ...
```

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

`putq()` — Put a Message on a Queue

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

`putq()` puts the message pointed at by *bp* on the message queue contained in the queue pointed at by *q* and enables that queue. `putq()` queues messages appropriately by type (i.e., message queueing priority, see the *Message Queues amd Service Procedures* section of the *STREAMS Module and Driver Programming* chapter).

`putq()` will always enable the queue when a Priority message is queued. `putq()` will enable the queue when an ordinary message is queued if the following condition is set, and enabling is not inhibited by `noenable()`: The condition is set if the module has just been pushed (see `I_PUSH` in `streamio`(4)), or if no message was queued on the last `getq()` call and no message has been queued since.

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

`qenable()` — Enable a Queue

```
int qenable(q)
    queue_t *q;
```

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

`qreply()` — Send Reverse-Direction Message on Stream

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

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

`qsize()` — Find the Number of Messages on a Queue

```
int qsize(q)
    queue_t *q;
```

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

`RD()` — Get Pointer to the Read Queue

```
#define RD(q) ...
```

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

`rmvb()` — Remove a Message Block from a Message

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

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

`rmvq()` — Remove a Message from a Queue

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

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

`splstr()` — Set Processor Level

```
int splstr()
```

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

`strlog()` — Submit Messages for Logging

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

`strlog()` submits messages containing specified information to the `log(4)` driver. Required definitions are contained in `<sys/strlog.h>` and `<sys/log.h>`. *mid* is the STREAMS module id number for the module or driver submitting the `log()` message. *sid* is an internal sub-id number usually used to identify a particular minor device of a driver. *level* is a tracing level that allows selective screening of messages from the tracer. *flags* are any combination of `SL_ERROR` (the message is for the error logger), `SL_TRACE` (the message is for the tracer), `SL_FATAL` (advisory notification of a fatal error), and `SL_NOTIFY` (request that a copy of the message be mailed to the system administrator). *fmt* is a `printf(3S)` style format string, except that %s, %e, %E, %g, and %G conversion specifications are not handled. Up to `NLOGARGS` numeric or character arguments can be provided. (See *Other Facilities* in the *Introduction to STREAMS* chapter of the *System Services Overview* and `log(4)`.)

`testb()` — Check for an Available Buffer

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

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

`unlinkb()` – Remove Message Block from Message Head

```
mblk_t *unlinkb(mp)
    mblk_t *mp;
```

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

`WR()` — Get Pointer to the Write Queue

```
#define WR(q) ...
```

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

A.5. Design Guidelines

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

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

General Rules

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

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

   The capability to pass `user` structure information upstream using messages has been provided where required. This can be done in `M_IOCTL` handling (see the *Drivers* section of the *STREAMS Module and Driver Programming* chapter and also *Message Types*, above. A module can send error codes upstream in a `M_IOCACK` or `M_IOCNAK` message, where they will be placed in `u_error` by the Stream head. Return values may also be sent upstream in a `M_IOCACK` message, and will be placed in `u_rval1`. Information can also be passed to the `user` structure via a `M_ERROR` message (see the *Complete Driver* section of the *STREAMS Module and Driver Programming* chapter and also *Message Types*, above. The Stream head will recognize this message type and inform the next system call that an error has occurred downstream by setting `u_error`. Note that in both instances, the downstream module cannot access the `user` structure, but it informs the Stream head to do so.

2. In general, modules should not require the data in an `M_DATA` message to follow a particular format, such as a specific alignment. This makes it easier to arbitrarily push modules on top of each other in a sensible fashion. Not

following this rule may limit module re-usability (the ability to use the module in multiple applications).

3. Every module must process an `M_FLUSH` message according to the value of the argument passed in the message. (See the *Message Queues and Service Procedures* and *Drivers* chapters of *STREAMS Module and Driver Programming*, and also *Message Types*, above.
4. A module should not change the contents of a data block whose reference count is greater than 1 (see `dupmsg()` in the *Utilities* section, above) because other modules that have references to the block may not want the data changed. To avoid problems, it is recommended that the module copy the data to a new block and then change the new one.
5. Modules should only manipulate message queues and manage buffers with the routines provided for those purpose, (see the *Utilities* section, above).
6. Filter modules pushed between a service user and a service provider (see the *Service Interface* section of the *STREAMS Module and Driver Programming* chapter) may not alter the contents of the `M_PROTO` or `M_PCPROTO` block in messages. The contents of the data blocks may be manipulated, but the message boundaries must be preserved.

System Calls

These rules pertain to module and drivers as noted.

1. *open* and *close* routines may sleep, but the sleep must return to the routine in the event of a signal. That is, if they sleep, they must be at priority <= `PZERO`, or with `PCATCH` set in the sleep priority.
2. The *open* routine must return >= 0 on success or `OPENFAIL` if it fails. This ensures that a failure will be reported to the user process. `errno` may be set on failure. However, if the open routine returns `OPENFAIL` and `errno` is not set, STREAMS will automatically set `errno` to `ENXIO`.
3. If a module or driver recognizes and acts on an `M_IOCTL` message, it must reply by sending a `M_IOCACK` message upstream. A unique id is associated with each `M_IOCTL`, and the `M_IOCACK` or `M_IOCNAK` message must contain the id of the `M_IOCTL` it is acknowledging.
4. A module (not a driver) must pass on any `M_IOCTL` message it does not recognize (see *Message Types*, above). If an unrecognized `M_IOCTL` reaches a driver, the driver must reply by sending a `M_IOCNAK` message upstream.

Data Structures

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

As described in the *SunOS STREAMS Topics* chapter, every module and driver is configured in with the address of a `streamtab` structure (see also the *Streams Mechanism* section of the *STREAMS Module and Driver Programming* chapter. For a driver, a pointer to its `streamtab` is included in `cdevsw`. For a module, a pointer to its `streamtab` is included in `fmodsw`.

Header Files

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

`types.h`
contains type definitions used in the STREAMS header files

`stream.h`
contains required structure and constant definitions

`stropts.h`
primarily for users, but contains definitions of the arguments to the `M_FLUSH` message type also required by modules

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

`errno.h`
defines various system error conditions, and is needed if errors are to be returned upstream to the user

`sysmacros.h`
contains miscellaneous system macro definitions

`param.h`
defines various system parameters, particularly the value of the `PCATCH` sleep flag

`signal.h`
defines the system signal values, and should be used if signals are to be processed or sent upstream

`file.h`
defines the file open flags, and is needed if `O_NDELAY` is interpreted

Accessible Symbols and Functions

The following lists the only symbols and functions that modules or drivers may refer to (in addition to those defined by STREAMS), if hardware and UNIX-system release independence is to be maintained. Drivers and modules which use symbols not listed here will not be compatible with System V systems.

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

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

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

```
short p_pid                 /* process ID */
short p_pgrp                /* process group ID */
```

Functions accessible from open/close procedures only

```
flg = sleep(chan, pri)    /* sleep until wakeup */
```

Universally accessible functions

```
bcopy(from, to, nbytes) /* copy data quickly */
bzero(buffer, nbytes)   /* zero data quickly */
t = max(a, b)           /* return max of args */
t = min(a, b)           /* return min of args */
mem=rm_alloc(map, size) /* allocate resource */
rmfree(map, size, addr) /* de-allocate resource */
rminit(mp, size, addr, name, mapsize)   /* initialize resource map */
printf(format, ...)     /* print message */
s = spln()          /* set priority level */
timeout(func, arg, ticks)/* schedule event */
untimeout(func, arg)    /* cancel event */
wakeup(chan)            /* wake up sleeper */
```

sysmacros.h

```
t = major(dev)          /* return major device */
t = minor(dev)          /* return minor device */
```

kernel.h

```
struct timeval boottime /* time since system came up */hz
struct timeval time     /* current time */
```

param.h

```
PZERO                   /* zero sleep priority */
PCATCH                  /* catch signal sleep flag */
hz                      /* clock ticks per second */
NULL                    /* 0 */
```

types.h

```
dev_t                   /* combined major/minor device */
time_t                  /* time counter */
```

All data elements are software read-only except:

```
u_error                 /* may be set on a failure return of open */
```

Rules for Put and Service Procedures

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

1. A put procedure must not sleep.
2. Each QUEUE must define a put procedure in its `qinit` (see *Kernel Structures*, above) structure for passing messages between modules.
3. A put procedure must use the `putq()` (see *Utilities*, above) utility to enqueue a message on its own message queue. This is necessary to ensure that the various fields of the `queue_t` structure are maintained consistently.
4. When passing messages to a neighbor module, a module may not call `putq()` directly, but must call its neighbor's put procedure (see

`putnext()` in *Utilities*) Note that this rule is distinct from the one above it. The previous rule states that a module must call `putq()` to place messages on its own message queue, whereas this rule states that a module must not call `putq()` directly to place messages on a neighbor's queue.

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

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

Service procedures must observe the following rules:

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

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

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

Step 2:
If all the following conditions are met:

- `canput()` fails and
- the message type is not a priority type (see *Message Types*) and
- the message is to be put on the next QUEUE.

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

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

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

A.6. STREAMS Glossary

Back Enable	To enable (by STREAMS) a preceding blocked QUEUE when STREAMS determines that a succeeding QUEUE has reached its low *water mark.*
Blocked	A QUEUE that cannot be enabled due to *flow control.*
Clone Device	A STREAMS device that returns an unused minor device when initially opened, rather than requiring the minor device to be specified in the `open(2)` call.
Close Procedure	The module routine that is called when a module is popped from a Stream and the driver routine that is called when a *driver* is closed.
Control Stream	In a *multiplexor*, the *upper Stream* on which a previous `I_LINK ioctl` (to the associated file, see `streamio`(4)) caused a *lower Stream* to be connected to the multiplexor driver at the end of the *upper Stream.*
Downstream	The direction from *Stream head* towards *driver.*
Device Driver	In the STREAMS context, the term "device driver" refers to the end of the *Stream* closest to an external interface. The principle functions of a *device driver* are handling an associated physical device, and transforming data and information between the external interface and *Stream.*
Driver	A module that forms the *Stream end.* It can be a *device driver* or a *pseudo-device driver.* In STREAMS, a *driver* is physically identical to a module (i.e., composed of two QUEUEs), but it has additional attributes.
Enable	Schedule a QUEUE.
Flow Control	The STREAMS mechanism that regulates the flow of messages within a Stream and the flow from user space into a Stream.
Lower Stream	A Stream connected below a multiplexor *pseudo-device driver*, by means of an `I_LINK ioctl`. The far end of a lower Stream terminates at a *device driver* or another multiplexor driver.
Message	One or more linked *message blocks.* A message is referenced by its first message block and its type is defined by the *message type* of that block.
Message block	Carries data or information, as identified by its *message type*, in a Stream. A *message block* is a triplet consisting of a data buffer and associated control structures, an `mblk_t` structure and a `dblk_t` structure.

Message Queue	A linked list of zero or more *messages* connected to a QUEUE.
Message type	A defined set of values identifying the contents of a *message block* and *message.*
Module	A pair of QUEUEs. In general, module implies a *pushable* module.
Multiplexor	A STREAMS mechanism that allows messages to be routed among multiple Streams in the kernel. A multiplexor includes at least one multiplexing *pseudo-device driver* connected to one or more *upper* Streams and one or more *lower* Streams.
Open Procedure	The routine in each STREAMS *driver* and *module* called by STREAMS on each `open(2)` system call made on the Stream. A *module*'s open procedure is also called when the *module* is pushed.
Pop	A STREAMS `ioctl()` (see `streamio`(4)) that causes the *pushable module* immediately below the *Stream head* to be removed (popped) from a Stream (modules can also be popped as the result of a `close(2)`) .
Pseudo-device Driver	A software *driver,* not directly associated with a physical device, that performs functions internal to a *Stream* such as a *multiplexor* or log *driver.*
Push	A STREAMS `ioctl()` (see `streamio`(4)) that causes a *pushable module* to be inserted (pushed) in a Stream immediately below the *Stream head.*
Pushable Module	A module interposed (pushed) between the *Stream head* and *driver.* Pushable modules perform intermediate transformations on messages flowing between the *Stream head* and *driver.* A *driver* is a non-pushable module and a *Stream head* includes a non-pushable module.
Put Procedure	The routine in a QUEUE which receives messages from the preceding QUEUE. It is the single entry point into a QUEUE from a preceding QUEUE. The procedure may perform processing on the message and will then generally either queue the message for subsequent processing by this QUEUE's *service procedure,* or will pass the message to the *put procedure* of the following QUEUE.
QUEUE	A STREAMS defined set of C-language structures. A module is composed of a read (*upstream*) QUEUE and a write (*downstream*) QUEUE. A QUEUE will typically contain a put and service procedure, a *message queue,* and private data. The read QUEUE (cf. *read queue*) in a *module* will also contain the open procedure and close procedure for the *module.* The primary structure is the `queue_t` structure, occasionally used as a synonym for a QUEUE.
Read Queue	The *message queue* in a *module* or *driver* containing *messages* moving *upstream.* Associated with a `read(2)` system call and input from a *driver.*
Schedule	Place a QUEUE on the internal list of QUEUEs which will subsequently have their service procedure called by the STREAMS scheduler.
Service Interface	A set of primitives that define a service at the boundary between a *service user* and a *service provider* and the rules (typically represented by a state machine) for allowable sequences of the primitives across the boundary. At a Stream/user boundary, the primitives are typically contained in the control part of a message;

	within a Stream, in `M_PROTO` or `M_PCPROTO` message blocks.
Service Procedure	The routine in a QUEUE which receives messages queued for it by the *put procedure* of the QUEUE. The procedure is called by the STREAMS *scheduler*. It may perform processing on the message and will generally pass the message to the *put procedure* of the following QUEUE.
Service Provider	In a *service interface*, the entity (typically a *module* or *driver*) that responds to request primitives from the *service user* with response and event primitives.
Service User	In a *service interface*, the entity that generates request primitives for the *service provider* and consumes response and event primitives.
Stream	The kernel aggregate created by connecting STREAMS components, resulting from an application of the STREAMS mechanism. The primary components are the *Stream head*, the *driver*, and zero or more pushable *modules* between the *Stream head* and *driver*.
Stream End	The end of the *Stream* furthest from the user process, containing a *driver*.
Stream Head	The end of the *Stream* closest to the user process. It provides the interface between the *Stream* and the user process.
STREAMS	A kernel mechanism that supports development of network services and data communication *drivers*. It defines interface standards for character input/output within the kernel, and between the kernel and user level. The STREAMS mechanism comprises integral functions, utility routines, kernel facilities and a set of structures.
Upper Stream	A Stream terminating above a multiplexor *pseudo-device driver*. The far end of an upper Stream originates at the *Stream head* or another multiplexor driver.
Upstream	The direction from *driver* towards *Stream head*.
Water Marks	Limit values used in *flow control*. Each QUEUE has a high water mark and a low water mark. The high water mark value indicates the upper limit related to the number of characters contained on the *message queue* of a QUEUE. When the enqueued characters in a QUEUE reach its high water mark, STREAMS causes another QUEUE that attempts to send a message to this QUEUE to become *blocked*. When the characters in this QUEUE are reduced to the low water mark value, the other QUEUE will be unblocked by STREAMS.
Write queue	The *message queue* in a *module* or *driver* containing *messages* moving *downstream*. Associated with a `write(2)` system call and output from a user process.

PART THREE: Non-STREAMS Appendices

B

Summary of Device Driver Routines

Summary of Device Driver Routines **351**

B.1. Standard Error Numbers 351

B.2. Device Driver Routines 351

- *xx*`attach()` — Attach a Slave Device 352
- *xx*`close()` — Close a Device 352
- *xx*`intr()` — Handle Vectored Interrupts 352
- *xx*`ioctl()` — Miscellaneous I/O Control 353
- *xx*`mmap()` — Mmap a Page of Memory 355
- *xx*`minphys()` — Determine Maximum Block Size 355
- *xx*`open()` — Open a Device for Data Transfers 356
- *xx*`poll()` — Handle Polling Interrupts 357
- *xx*`probe()` — Determine if Hardware is There 357
- *xx*`read()` — Read Data from Device 358
- *xx*`select()` — Select Support 358
- *xx*`strategy()` — High-Level I/O 359
- *xx*`write()` — Write Data to Device 359

B

Summary of Device Driver Routines

B.1. Standard Error Numbers

The system has a collection of standard error numbers that a driver can return to its callers. These numbers are described in detail in `intro(2)`, the introductory pages of the *System Interface Manual*. A complete listing of the error numbers appears in `<sys/errno.h>`.

B.2. Device Driver Routines

These routines actually compose the bulk of the device driver. Some of them, like `xxioctl()`, are optional. Others, like `xxprobe()`, must appear in every driver. Omitted from this section is the `xxslave()` routine, which appears primarily in block-device drivers. See the *The "Skeleton" Character Device Driver* chapter for additional information about many of these routines.

When a user program makes a system call that involves I/O devices, it's translated by the kernel into a call to the appropriate driver routine. However, when that driver routine is called, its parameters are no longer the same as the parameters that the user program passed to the system call — they will have been translated into parameters reflecting the actual run-time environment of the drivers, an environment set up and initialized by `config` and the autoconfiguration process and then maintained by the kernel and the drivers themselves. For example, a user program will call

```
write (fileno, address, nbytes)
    int fileno;
    char *address;
    int nbytes;
```

but the kernel will translate this into

```
xxwrite(dev, uio)
    dev_t dev;
    struct uio *uio;
```

by the time it calls the driver's `xxwrite()` routine.

_xx_attach() — Attach a Slave Device

```
xxattach(md)
    struct mb_device *md;
```

_xx_`attach()` does boot-time, device-specific setup and initialization. It's commonly used in disk and tape drivers for setup tasks like reading labels, and in character drivers for the initialization of interrupt vectors and the reserving of blocks of memory. Its proper tasks are not limited to the initialization of actual hardware devices — _xx_`attach()` is also used to set up and initialize local data structures.

When it needs to set a device interrupt-vector number, _xx_`attach()` finds it in the `md_intr->v_vec` field of the `mb_device` structure. A `NULL` value in this field indicates that the host machine is Multibus based and does not support vectored interrupts. On VMEbus machines `md_intr->v_vec` is the interrupt-vector number given for the device in the kernel config file and *must* be present.

_xx_`attach()` can also be used to set the 32-bit argument that's subsequently passed to _xx_`intr()`. This argument (contained in `md_intr->v_vptr` is initially set to the unit number of the interrupting device, but it's often convenient to reset it to contain a pointer to a local structure.

_xx_close() — Close a Device

```
xxclose(dev, flags)
    dev_t dev;
    int flags;
```

_xx_`close()` does whatever it has to do to indicate that data transfers can't be made on the device until it's been reopened. This may involve nothing at all, or it may include resetting and quieting the device, flushing data buffers, and releasing or unlocking resources (or unlocking the device itself if it's opened exclusively). Since _xx_`close()` is called only when the *last* user process which is using the device closes it, _xx_`close()` must clean up for *all* user processes which have had the device open. _xx_`close()` doesn't need to report an error, although it can. *flags*, incidently, is the same as it is for _xx_`open()`.

_xx_intr() — Handle Vectored Interrupts

```
xxintr(ctrl_num)
    int ctrl_num;
```

_xx_`intr()` is responsible for fielding vectored interrupts from the device. As such, it is specified (with its interrupt vector) in the kernel config file. As an interrupt routine, _xx_`intr()` (and any routines that it calls) is absolutely prohibited from calling `sleep()` or referencing the kernel `user` structure.

_xx_`intr()` receives one 32-bit parameter, which is, by default, the unit number of the device that interrupted. However, you can arrange for it to receive something else by changing the value in `md_intr->v_vptr`. (See *xx*`attach()`, above).

In character drivers which, like block drivers, make use of `physio()` and its associated structures, mechanisms and routines, *xx*`intr()` is used to indicate when the device is finished with one chunk and ready for the next. *xx*`intr()` is also instrumental in certain tasks which, by their nature, must be shared with top-half routines. Examples of such tasks are the maintenance of character I/O buffers and `select()`-related bookkeeping structures. (In the `select()` case, *xx*`intr()` also has the job of calling `selwakeup()` to wakeup sleeping processes).

Note that whenever *xx*`intr()` maintains a data structure or resource in cooperation with top-level routines, the top-level code *must* be protected by a mutual-exclusion lock. Interrupts are automatically disabled when an interrupt routine is called, so it is generally unnecessary for *xx*`intr()` to disable interrupts before it does its part of the job.

xx`intr()` is also responsible for error handling and reporting. More specifically:

- *xx*`intr()` should check the device for an error every time it's called. It can also check the driver state against the device state to ensure that the device is, in fact, doing what the driver expects it to be doing. Upon finding an "impossible" or unrecoverable error, *xx*`intr()` should `panic()`. But for regular errors it should call `printf()` (or `uprintf()`), flag the error in the I/O buffer, and then return.
- The error is flagged by setting the `B_ERROR` bits in the buffer header `b_flags` field (and, if an error code other than `EIO` is desired, by assigning that error code into the buffer `b_error` field). The error code will then be propagated up to the user by way of `physio()`. `physio()` checks to see if the error flag has been set in the buffer, and if it has, passes the error code up to the user program, which usually plugs it into the global error register `errno`. *xx*`intr()` doesn't itself return anything.

Note that the driver *xx*`intr()` routine cannot itself set the `errno` register, since that register is actually a field in the `user` structure (`u.u_error`), and the `user` structure *must not* be accessed at interrupt time. Instead, *xx*`intr()` passes the error to the kernel via the buffer, and the kernel sets `u.u_error`.

- A retry attempt can be made before giving up and taking the error return. Whether or not this is advisable is entirely dependent on the specific device and error characteristics. (Note that the `b_resid` field in the buffer header will typically indicate the number of bytes of data that were still untransferred at the error return).
- The error return should abort the I/O request that produced the error and then place the device in its normal idle state.

xx`ioctl()` — Miscellaneous I/O Control

```
xxioctl(dev, cmd, data, flag)
    dev_t dev;
    int cmd;
    caddr_t data;
    int flag;
```

The device-driver entry routines, taken as a set, are intended to constitute a uniform abstract interface capable of accommodating all possible I/O devices. Obviously, such devices differ greatly, and thus the need for this *xx*`ioctl()`. It is the escape mechanism by which miscellaneous operations are accommodated.

These functions vary greatly — almost anything is possible. The range of possibilities requires a very general interface, and *xx*`ioctl()` has one. The *cmd* variable identifies a specific device control operation, and is typically used by *xx*`ioctl()` as the index into a switch statement. The *data* parameter is the real escape hatch, a pointer to an array up to 255 bytes in length. This array, over which the driver and its users will overlay a driver-specific structure, can be treated as both an input parameter by which user programs send data to the driver and as an output parameter by which the driver returns data to its users. *flag* is set to the `f_flags` field of the `file` structure. The `file` structure, together with the file-mode flags to which its `f_flags` field can be set (`FREAD`, `FWRITE`, and so on) is defined in `<sys/file.h>`. The driver is free to use *flag* to make its operation sensitive to the manner in which the file was opened by the user.

In `<sys/ioctl.h>` will be found a collection of macros which encode parameter size and read/write control information into `ioctl()` command codes. These macros tell the kernel, on a command by command basis:

- How many of the maximum of 127 bytes in the `ioctl()` parameter are significant when that parameter is read.
- How many of these bytes are significant when the parameter is written.
- If the parameter bytes should be read into kernel space before calling *xx*`ioctl()`.
- If they should be read into user space after calling *xx*`ioctl()`.

The Versatec Interface driver in the *Sample Driver Listings* appendix of this manual contains some simple examples of the use of these `ioctl()` macros. (More complex examples can be found in `<sys/ioctl.h>`). The Versatec Interface driver defines two `ioctl()` command codes (in `/usr/include/sys/vcmd.h`):

```
#define   VGETSTATE _IOR(v, 0, int)
#define   VSETSTATE _IOW(v, 1, int)
```

The first parameter of the `ioctl()` macros is an ASCII character that serves to group together each driver's command codes. It must be different for each device — in this case, it's "v" for "Versatec". The second parameter is the command code itself. The third is the size of the `ioctl()` argument, which cannot exceed 127 bytes. Note that the size is given as the name of the structure which will be used to interpret the parameter array. The macros `_IOR`, `_IOW` and `_IOWR` then use the `sizeof()` operator to determine the number of bytes consumed by the structure.

The definitions of such `ioctl()`-related structures, together with the command-code definitions themselves, must be collected into a user accessible include file. Such include files are usually, though not necessarily, kept in

`/usr/include/sys`.

When the kernel processes the `ioctl()` system call, translating its parameters into the terms appropriate to an *xx*`ioctl()` driver routine, it consults the read/write encode bits in the command code. If the read bit is set, then the argument is read into a buffer in kernel space, and a pointer to that buffer is passed to the driver `ioctl()` routine. Likewise, if the write bit is set, the argument is copied back into user space after command execution is completed.

xx`ioctl()` does whatever it has to do, then returns 0 if there were no errors, an error code if there were. `ENOTTY` is the code used if the requested command did not apply to the device. The kernel passes error codes up to the user program, which usually plugs them into `errno`.

*xx*mmap() — Mmap a Page of Memory

```
xxmmap(dev, off, protection)
    dev_t dev;
    off_t off;
    int protection;
```

*xx*mmap`()` is called for PTE information about the page (at offset `off`) of *dev*'s memory. (This information is what the kernel needs to map the page to a virtual address). *xx*mmap`()` should first check that `off` doesn't exceed the device-memory size:

```
if (off >= XXSIZE) return (-1);
```

for this would cause the mapping of an area greater than the device memory. *xx*mmap`()` returns the subset of the page table entry (PTE) containing the page frame number and the page type to its caller in the kernel. *xx*mmap`()` is called iteratively to perform a mapping requested by a call to mmap`()` — the looping and all of its bookkeeping, as well as the actual mapping, is performed by the kernel in a way that's transparent to the driver.

*xx*mmap`()` returns -1 to the kernel if it can't do the mapping, otherwise it returns its PTE subset. Upon receipt of a -1, the kernel returns the error code `EINVAL` (Illegal argument) to the user program, where it's usually plugged into the global error variable `errno`.

*xx*minphys() — Determine Maximum Block Size

```
unsigned xxminphys(bp)
    register struct buf *bp;
```

xx`minphys()` determines a "reasonable" block size for transfers, so as to avoid tying up too many resources. *xx*`minphys()` is passed as an argument to `physio`. The system version of the *xx*`minphys()` function, `minphys`, may be used by any driver. *xx*`minphys()` should perform the calculation:

```
int block; /* some reasonable block sixe for transfers */

if (bp->b_bcount > block)
    bp->b_bcount = block;
```

xxopen() — Open a Device for Data Transfers

```
xxopen(dev, flags)
    dev_t dev;
    int flags;
```

xx`open()` is called each time the device is opened, and may include any device-specific initialization. Typically, it will:

- begin by validating the minor device number and doing other device-specific error checking.
- Then if everything is ok, it will initialize the device (for example by clearing registers, enabling interrupts or checking for power-up errors) and possibly the local data structures. This structure initialization may include locking the device if it's exclusive use, or allocating driver resources — for example allocating dynamic buffers that will be needed later).
- Finally, *xx*`open()` will typically wait for the device to come on-line, and return an error if it doesn't.

NOTE *If xx*`open()` supports "clone open", that is to say, if it will allow a user to open a driver without specifying a minor device, then it is important that it does anything that may lead to its being blocked before it actually chooses the minor device that it is going to clone. Otherwise, there's a possibility of someone else grabbing the device while *xx*`open()` is blocked.

The integer argument *flags* indicates if the open is for reading, writing, or for both. The constants `FREAD` and `FWRITE` (from `<sys/file.h>`) are available to be AND'ed with *flags*.

The minor device number encoded in *dev* is of concern only to the device driver itself. It can itself be encoded to contain various kinds of information, as needed by the driver. The driver developer will want to provide macros to break out encoded subfields. *dev* may encode a unit or driver number, a special feature, or an operating mode.

xx`open()` returns `ENXIO` (No such device or address) if the minor device number is out of range, `ENODEV` (No such device) if an attempt was made to open the device with an inappropriate mode or `EIO` (I/O Error) to indicate an I/O error in the course of an attempted initialization. If the open is successful, *xx*`open()` returns 0. The kernel will return the error code to the user program, where it is usually plugged into the global error variable `errno`.

xx`poll()` — Handle Polling Interrupts

```
xxpoll()
```

xx`poll()` is responsible for fielding non-vectored interrupts from the device. In situations where multiple devices share the same interrupt level, *xx*`poll()` must determine if the interrupt was actually destined for this driver or not. *xx*`poll()` returns 0 to indicate that the interrupt was not serviced by this driver, and non-zero to indicate that the interrupt was serviced. It is a gross error for *xx*`poll()` to say that it serviced an interrupt when it did not.

If a device driver handles both vectored interrupts and polling interrupts, *xx*`poll()` typically calls the *xx*`intr()` routine with the proper arguments, normally the unit number of the device that interrupted. `sleep` may never be called from *xx*`poll()`, or, for that matter, from any of the lower-half routines.

xx`probe()` — Determine if Hardware is There

```
xxprobe(reg, unit)
    caddr_t reg;
    int unit;
```

xx`probe()` determines whether the device at the kernel virtual address *reg* actually exists and is the correct device for this driver. The method by which it accomplishes this is impossible to standardize, for devices provide no uniform means of identification. Indeed, some devices fail to provide even reasonable non-standard means of identification.

The kernel provides a set of functions to help with probing. These functions can probe an address, recover from the bus error that will occur if no device is installed at that address, and return with an indication as to whether such a bus error occurred. These functions are `peek()`, `peekc()`, `peekl()`, `poke()`, `pokec()` and `pokel()`.

It's possible for `probe()` to check the value of the *reg* parameter to ensure that the device isn't installed at an address that it can't itself address. The device's entry in the kernel config file determines which address space it's mapped into, but it's sometimes possible for the device itself to be configured differently. The driver can check, for example, that *reg* doesn't contain an address greater than `0xFFFFF` (that is, an address with more than 20 significant bits) if the device is configured for 20-bit references.

It's also possible for *xx*`probe()` to do some device initialization, even though such initialization is properly the job of *xx*`attach()`. This can make sense if such initialization allows *xx*`probe()` to identify and verify the device, but it should only do the amount of initialization necessary to determine if the device is really there. It definitely should not allocate any memory that won't be used if the device isn't found, and it should not assume that just because it found a device that the system will choose to include that device in its configuration

If the correct device is found at the probed location, *xx*`probe()` returns (sizeof (struct *xx*`device()`)). (This is the size of the device registers in I/O space if the device is an I/O mapped Multibus device; otherwise it's the size of the device registers in memory space). If no device is found at the expected location, or if the device found is not the one that was expected, *xx*`probe()` returns a 0. If it doesn't, the kernel will be incorrectly led to believe that a device is present, and future attempts to contact it will cause the kernel to `panic()` with a bus error.

Note that the amount of memory mapped in by the autoconfiguration code is determined by the size given in the `mb_driver->mdr_size` field, and not by the value returned from *xx*`probe()`, which is used only for the go/nogo test.

xx`read()` — Read Data from Device

```
xxread(dev, uio)
    dev_t dev;
    struct uio *uio;
```

xx`read()` is the high-level routine called (in character device drivers) to perform data transfers from the device. *xx*`read()` must check that the minor device number passed to it is in range. If the minor device number is out of range, *xx*`read()` returns like so:

```
if (XXUNIT(dev) >= NXX)
    return (ENXIO);
```

Subsequent actions of *xx*`read()` differ depending on whether the device is a tty-style character-at-a-time device or a device that buffers its I/O into blocks.

For block transfers, *xx*`read()` uses `physio()`, its associated mechanisms, and the *xx*`strategy()`. `buf` is here an array of locally declared buffers:

```
return (physio(xxstrategy, &buf[minor(dev)],
    dev, B_READ, minphys, uio));
```

If the read operation fails, *xx*`read()` passes the error code which *xx*`intr()` set in the buffer header up to the kernel. The kernel then passes it on to the user program, which usually plugs it into the global error variable `errno`.

xx`select()` — Select Support

```
xxselect(dev, rw)
    dev_t dev;
    int rw;
```

The *xx*`select()` routine is necessary if the driver is to support the `select()` system call. *rw* is either `FREAD`, `FWRITE` or 0. (Simple character devices won't have occasion to use the 0 value, which is intended for exceptional conditions. It is used by network devices). These constants are defined in `<sys/file.h>`.

If *xx*`select()` only supports polling, then it simply determines if the device specified by (the major/minor pair encoded within) `dev` is ready to go, returning a 1 if it is and a 0 if it's not. Interrupts must be disabled while this check is performed, so *xx*`select()` should always do a

```
s = spl5();
```

immediately, and a

```
splx(s)
```

before returning.

If, however, *xx*`select()` allows user processes to wait for a device to become ready, it must do somewhat more work. In this case, the driver will have to maintain a local per-device structure which can associate a process with each device. It can do so with the current process `proc` structure, a pointer to which can be found in `u.u_procp`. (If the device can read and write independently, separate processes must be tracked for the two cases). The local structures must also contain some state information, which will be used by *xx*`select()` (as well as *xx*`intr()`) for bookkeeping purposes. The details are somewhat complicated, and are illustrated in the *Variation with "Asynchronous I/O" Support* section of the *The "Skeleton" Character Device Driver* chapter of this manual.

xx`strategy()` — High-Level I/O

```
xxstrategy(bp)
    register struct buf *bp;
```

xx`strategy()` is a high-level I/O routine designed to be called from `physio()`. Its name derives from its role in block-device drivers, where *xx*`strategy()` has responsibility for reordering the I/O request queue so as to increase the overall I/O bandwidth. In character devices (even those which queue I/O) such reordering is to no advantage, and *xx*`strategy()`'s major function is structural. It allows the *xx*`read()` and *xx*`write()` routines to share their common code in a routine designed to be called from `physio()`. *xx*`strategy()` returns no error code to its caller in the kernel. Instead, errors that occur in the course of the I/O operation are reported by *xx*`intr()` by way of the buffer header and passed along by *xx*`strategy()`.

xx`write()` — Write Data to Device

```
xxwrite(dev, uio)
    dev_t dev;
    struct uio *uio;
```

xx`write()` is the high-level routine called (in character device drivers) to perform data transfers to the device. *xx*`write()` must check that the minor device number passed to it is in range. If the minor device number is out of range, *xx*`write()` returns like so:

```
if (XXUNIT(dev) >= NXX)
    return (ENXIO);
```

Subsequent actions of *xx*`write()` differ depending on whether the device is a tty-style character-at-a-time device or a device that buffers its I/O into blocks.

For block transfers, *xx*`write()` uses `physio()`, its associated mechanisms, and the *xx*`strategy()`. `buf` is here an array of locally declared buffers:

```
return (physio(xxstrategy, &buf[minor(dev)],
    dev, B_WRITE, minphys, uio));
```

If the write operation fails, *xx*`write()` passes the error code which *xx*`intr()` set in the buffer header up to the kernel. The kernel then passes it on to the user program, which usually plugs it into the global error variable `errno`.

C

Kernel Support Routines

Kernel Support Routines 363

`btodb()` — Convert Bytes to Disk Sectors 363
`copyin()` — Move Data From User to Kernel Space 363
`copyout()` — Move Data From Kernel to User Space 363
`CDELAY()` — Conditional Busy Wait 364
`DELAY()` — Busy Wait for a Given Period 364
`dma_done()` — Free the DMA Channel 364
`dma_setup()` — Set Up for a DMA Transfer 364
`gsignal()` — Send Signal to Process Group 368
`hat_getkpfnum()` — Address to Page Frame Number 368
`inb()` — Read a Byte from an I/O Port 368
`iodone()` — Indicate I/O Complete 369
`iowait()` — Wait for I/O to Complete 369
`kmem_alloc()` — Allocate Space from Kernel Heap 369
`kmem_free()` — Return Space to Kernel Heap 369
`log()` — Log Kernel Errors 370
`MBI_ADDR()` — Get Address in DVMA Space 370
`mapin()` — Map Physical to Virtual Addresses 370
`mapout()` — Remove Physical to Virtual Mappings 372
`mbrelse()` — Free Main Bus Resources 372
`mbsetup()` — Set Up to Use Main Bus Resources 372
`outb()` — Send a Byte to an I/O Port 373
`panic()` — Reboot at Fatal Error 373

peek(), peekc(), peekl() — Check and Read 373
physio() — Block I/O Service Routine 373
poke(), pokec(), pokel() — Check and Write 375
printf() — Kernel Printf Function 376
pritospl() — Convert Priority Level 376
psignal() — Send Signal to Process 377
rmalloc() — General-Purpose Resource Allocator 377
rmfree() — Recycle Map Resource 378
selwakeup() — Wakeup a Select-blocked Process 378
sleep() — Sleep on an Event 378
spl*n*() — Set CPU Priority Level 379
splx() — Reset Priority Level 379
suser() — Reset Priority Level 380
swab() — Swap Bytes 380
timeout() — Wait for an Interval 380
uiomove() — Move Data To or From an **uio** Structure 380
untimeout() — Cancel **timeout** Request 381
uprintf() — Nonsleeping Kernel Printf Function 381
ureadc(), uwritec() — uio Structure Read/Write 381
wakeup() — Wake Up a Process Sleeping on an Event 382

C

Kernel Support Routines

These routines are in alphabetical order, on the assumption that this will make them easier to find than any "logical" order.

`btodb()` — Convert Bytes to Disk Sectors

```
btodb(bytes)
    int bytes;
```

Converts *bytes* into standard kernel block-size units. `btodb()` is called (for block drivers) from *xx*`size()`. It is listed here because it is called from the example ramdisk pseudo-device driver.

`copyin()` — Move Data From User to Kernel Space

`copyin()` moves data from the user address space to the kernel address space. It is commonly used when writing *xx*`ioctl()` routines. See `copyout()`.

```
copyin(udaddr, kaddr, n)
    caddr_t udaddr, kaddr;
    u_int n;
```

where *kaddr* is a kernel virtual address, *udaddr* is a user virtual address, and *n* is the number of bytes to copy in. Returns 0 if no error occurs, `EFAULT` on a memory error, and other `Exxx` errors on pagefaults which cannot be resolved.

`copyout()` — Move Data From Kernel to User Space

`copyout()` moves data from the kernel address space to the user address space. It is commonly used when writing *xx*`ioctl()` routines. See `copyin()`.

```
copyout(kaddr, udaddr, n)
    caddr_t kaddr, udaddr;
    u_int n;
```

where *kaddr* is a kernel virtual address, *udaddr* is a user virtual address, and *n* is the number of bytes to copy out. Returns 0 if no error occurs, `EFAULT` on a memory error, and other `Exxx` errors on pagefaults which cannot be resolved.

CDELAY() — Conditional Busy Wait

```
CDELAY(condition, time)
    int condition, time;
```

`CDELAY()` is like `DELAY()` (see below) in that it busy waits for a specified number of microseconds. It differs, however, in that it has a second argument *condition*. Each time it goes through its busy wait loop, `CDELAY()` checks *condition*, and, if it's true, it immediately returns. In typical usage, *condition* is a masked subset of the bits in a device register.

DELAY() — Busy Wait for a Given Period

```
DELAY(time)
    int time;
```

`DELAY` busy waits for a specified minimum number of microseconds. That is, it just spins around using CPU time. It can be useful in situations where a device is not quite slow enough to justify having its driver go to sleep. In such cases, it's useful to busy wait for a short time. The reasoning is that while busy waiting is a waste, servicing an interrupt costs a lot more CPU time.

`DELAY()` is also useful in introducing pauses between accesses to a device with write latency. A device register may, for example, require multiple sequential writes, and yet also require delays between the writes. See `vpprobe` in the *Sample Driver Listings* appendix for an example. See `CDELAY()`.

dma_done() — Free the DMA Channel

```
dma_done(chan)
    int chan;
```

On Sun386i only. After a DMA transfer completes, `dma_done()` must be called to mark the channel as not busy so that another transfer can proceed.

dma_setup() — Set Up for a DMA Transfer

```
dma_setup(dma)
    struct dma_request *dma;
```

On Sun386i only. `dma_setup()` is called after the driver has gotten a contiguous set of virtual addresses from `mbsetup()` and before the device is programmed to start sending or receiving data. The `dma_request` structure (defined in `/usr/include/sun386/dma.h`) contains all the information required to set up the 82380 DMA chip on the Sun386i.

Unlike the Sun-2, Sun-3, Sun-4 line of machines, the Sun386i has a memory management unit as an integral part of the CPU (the 80386). Therefore, to use the DMA facility of the Sun386i for a device driver, you must interface to the 82380 chip, which contains the DMA controller.

The primary interface to the DMA chip is the `dma_request` structure. You must fill in the fields in this structure and then call `dma_setup()` with a pointer to the structure. `dma_setup()` takes the contiguous virtual addresses, which were obtained from a call to `mbsetup()`, and sets up a linked list of physical addresses to be loaded into the DMA chip as needed.

`dma_setup()` returns a value of zero if the setup was successful, and non-zero if there is a problem. Reasons for failure are: the channel was busy, the transfer was zero pages long, or memory could not be allocated for the linked list of buffers.

The fields in `dma_request` structure are defined as follows:

```
/*
 * DMA request structure passed to dma_setup().
 * See the Intel 82380 Tech Ref for more info.
 */
struct dma_request {
    u_char  dma_channel;           /* Channel number: 0 - 7 */
    u_char  dma_xfer_mode;         /* Transfer mode */
#define    DMA_DEMAND_MODE     0
#define    DMA_SINGLE_MODE     1
#define    DMA_BLOCK_MODE      2
#define    DMA_CASCADE_MODE    3
    char    dma_rdwr;              /* Transfer direction */
#define    DMA_READ            2     /* (Relative to requester) */
#define    DMA_WRITE           1
    u_long  dma_count;             /* Transfer count */
    u_long  dma_req_space;         /* Requester address space */
#define    DMA_MEMORY          0     /* Memory or memory-mapped */
#define    DMA_IO              1     /* I/O mapped */
    u_int   dma_req_size;          /* Size of xfers to/from requester */
#define    DMA_BUS_32          1     /* 32-bit transfers */
#define    DMA_BUS_16          2     /* 16-bit transfers */
#define    DMA_BUS_8           3     /* 8-bit transfers */
    char    dma_req_hold;          /* 1 = hold address, 0 = increment *
    caddr_t dma_req_addr;          /* Requester (virtual) address */
    u_long  dma_target_space;      /* Target address space */
    u_int   dma_target_size;       /* Size of xfers to/from target */
    char    dma_target_hold;       /* Hold/increment target address */
    caddr_t dma_target_addr;       /* Target (virtual) address */
};
```

In this context, the "requester" is the device that requests service from the 82380 (normally a peripheral such as a disk controller). The "target" is the "device" with which the requester wants to communicate (normally system memory).

The fields of the `dma_request` structure are used as follows:

`dma_channel`
: Specifies the channel that the requester will use for the transfer.

`dma_xfer_mode`
: Refers to the type of transfer that the requester is capable of supporting. The SCSI controller, for instance, uses the `DMA_SINGLE_MODE` of transfer, as does the floppy controller. Refer to the peripheral manufacture's specification sheet and the the 82380 data sheet for more details.

`dma_rdwr`
: is the direction of data transfer *relative to the requestor.* `DMA_WRITE` means transfer from the requester to the target and `DMA_READ` means transfer from the target to the requester.

`dma_count`
: is the byte count for the transfer.

`dma_req_space`
: is the address space in which the requester resides, i.e., whether the device is memory mapped (`DMA_MEMORY`) or I/O mapped (.`DMA_IO`)

`dma_req_size`
: is the size of the requester's data path (`DMA_BUS_8` = 8 bits, `DMA_BUS_16` = 16 bits, `DMA_BUS_32` = 32 bits) and therefore the amount of data transferred with each DMA bus cycle.

`dma_req_hold`
: indicates whether the 82380 should hold the requester address constant throughout the DMA transfer, or increment it with each bus cycle. Typically the requester address is the address of the device's I/O register, which is fixed, so dma_req_hold is set to "1".

`dma_req_addr`
: is the requester's virtual address.

`dma_target_space`
: is the address space in which the target resides (usually `DMA_MEMORY`).

`dma_target_size`
: is the size of the target's data path (`DMA_BUS_32` for system memory).

`dma_target_hold`
: indicates whether the 82380 should hold or increment the target address during the DMA transfer. For memory devices, the 82380 should increment the target address with each bus cycle, so "dma_target_hold" is set to 0.

`dma_target_addr`
: is the target's virtual address.

Once all these fields are set up by the driver, the driver calls the `dma_setup()` routine. The following pseudo-code routines demonstrate how to use the DMA routines:

```
#include <machine/dma.h>
#include <sundev/mbvar.h>

struct   mb_device *xxinfo;          /* Device info */
caddr_t   xx_ioaddr = XX_ADDR;       /*Address of device's I/O port */

xx_example(bp)
   struct   buf *bp;
{
   struct   mb_device *md = xxinfo[0];
   unsigned int target_addr;
   unsigned int transfer_count;
   int   channel;
   int   readflag;

 /*
  * Set up DMA transfer.
  */
   target_addr = MBI_ADDR(mbsetup(md->md_hd, bp, 0));
   transfer_count = bp->b_bcount
   channel = md->md_dmachan;
   readflag = ((bp->b_flags & B_READ) ? 1 : 0);

   if (xx_dma_setup(target_addr, transfer_count,
     channel, readflag)  !=  0)
      return(-1);

 /*
  * Code to talk to the device, initiate the transfer,
  * and wait for transfer completion.
  */
    .
    .
    .

 /*
  * Free DMA resources.
  */
   xx_dma_done(channel);
   mbrelse(md->md_hd, &target_addr);

   return(0);
}

xx_dma_setup(addr, count, chan, rdflag)
  unsigned int addr;
  unsigned int count;
  int   chan;
  int   rdflag;
{
  struct   dma_request dreq;

  dreq.dma_channel = chan;               /* Dma channel */
  dreq.dma_xfer_mode =
    DMA_SINGLE_MODE;                     /* Single mode transfer */
  dreq.dma_rdwr =
    (rdflag ? DMA_WRITE : DMA_READ); /* Direction */
  dreq.dma_count = count;            /* Transfer count */

  dreq.dma_req_space = DMA_MEMORY;   /*Memory-mapped requester*/
  dreq.dma_req_size  = DMA_BUS_8;    /* 8-bit data path */
  dreq.dma_req_hold  = 1;            /* Hold address constant */
  dreq.dma_req_addr  = xx_ioaddr;    /* I/O port virt. address */

  dreq.dma_target_space = DMA_MEMORY; /*Target is system memory*/
  dreq.dma_target_size = DMA_BUS_32; /* 32-bit data path */
  dreq.dma_target_hold = 0;          /*Increment addr. each cycle*/
  dreq.dma_target_addr = addr;       /* Buffer virtual address */
  return(dma_setup(&dreq));
}

xx_dma_done(chan)
   int   chan;
{
   dma_done(chan);
}
```

gsignal() — Send Signal to Process Group

```
gsignal(pgrp, sig)
    int pgrp;
    int sig;
```

Sends signal *sig* to all of the processes in the process group *pgrp*. See `psignal()`.

hat_getkpfnum() — Address to Page Frame Number

```
unsigned int
hat_getkpfnum(addr)
    addr_t addr;
```

`hat_getkpfnum` takes a virtual address and returns it associated Page Frame Number. This number has already been masked down to one that can appropriately be returned by the driver *xx*`mmap()` routine.

inb() — Read a Byte from an I/O Port

```
inb(port)
    short port;
```

Sun386i only. `inb()` returns the byte value from the specified port address in the I/O space. (See `outb()`).

iodone() — Indicate I/O Complete

```
iodone(bp)
    struct  buf  *bp;
```

`iodone` is called to indicate that I/O associated with the buffer header *bp* is complete, and that it can be reused. `iodone` sets the `DONE` flag in the buffer header, then does a `wakeup` call with the buffer pointer as argument. `iodone()` is called from the bottom half right after the call to `wakeup()`. See `iowait()`.

iowait() — Wait for I/O to Complete

```
int iowait(bp)
    struct  buf  *bp;
```

`iowait` waits on the buffer header addressed by *bp* for the `DONE` flag to be set. `iowait` actually does a `sleep` on the buffer header and is called from the top half in place of `sleep()`. `iowait()` also returns the error value. See `iodone()`.

kmem_alloc() — Allocate Space from Kernel Heap

```
caddr_t kmem_alloc(nbytes)
    u_int nbytes;
```

Allocates *nbytes* of contiguous kernel memory and returns a pointer to it. If called from an interrupt routine, `kmem_alloc()` can return a `NULL`. (Though `kmem_alloc()` generally should not be called from the interrupt level.) It calls `panic()` if its request can't be satisfied. Note that `kmem_alloc()` takes a while, and shouldn't be used frivolously. (Also note that it can't, in system releases prior to 3.2, be called by `probe()` or `attach()`, since the kernel heap from which it allocates is not yet initialized). Memory allocated with `kmem_alloc()` can be recycled with `kmem_free()`.

kmem_free() — Return Space to Kernel Heap

```
kmem_free(ptr, nbytes)
    caddr_t ptr;
    u_int nbytes;
```

Returns the block (allocated by `kmem_alloc()`) at *ptr* to the kernel heap. If the block has already been freed, or if *ptr* doesn't indicate an address within the heap, `kmem_free()` panics. When the block is freed, it is coalesced with adjacent free blocks to ensure that the free blocks in the heap are as large as possible. `kmem_free()`, like `kmem_alloc()`, should not be called from the interrupt level.

`log()` — Log Kernel Errors

```
log(pri_code, . . .)
    int pri_code;
    . . .
```

The kernel provides a `log()` function analogous to the `syslog(3)` function supplied with the C library for user programs. The first argument to `log()` is a priority code, as defined in `<sys/syslog.h>`, and is identical to the priority codes used by `syslog(3)`. The subsequent arguments are a `printf()` format string and the values to be printed under its control. Unlike `syslog()`, the format string must be terminated with a newline (\n) if a newline is to be printed at the end of the message.

Messages logged with `log()` will not pass though the normal kernel `printf()` mechanism if the `syslogd` daemon is running. They will get written to the system message buffer just as `printf()` messages are. The `syslogd` daemon will read them using a special device driver, and will log them as messages from the ''kern'' facility with the given priority.

If such a message is to be printed on the console, `syslogd` will do so, using its standard format which includes a time stamp. Messages printed with `printf()` will get logged as messages from the ''kern'' facility with a priority of `LOG_CRIT`, except that `syslogd` will not print them on the console as they have already been printed there by the kernel. The kernel does not time stamp messages that it prints; thus, messages logged with `log()` will be time stamped if they are printed on the console, while messages printed with `printf()` will not. Furthermore, `syslogd` does not lock out interrupts while printing messages, so messages logged with `log()` will not tie up the machine while they are being printed, unless `syslogd` is not printing and the kernel must print the message itself.

`MBI_ADDR()` — Get Address in DVMA Space

```
MBI_ADDR(mb_cookie)
    int mb_cookie;
```

`MBI_ADDR()` is a macro that takes the ''cookie'' (abstract number) returned by `mbsetup()` and converts it into a 32-bit transfer address, which may be either in the DVMA space or a VMEbus address space. This is the address that is then given to the bus-master device, though it may first need to be checked (especially for older devices) to ensure that it is not larger than the device capacity. See `mbsetup()` and `mbrelse()`.

`mapin()` — Map Physical to Virtual Addresses

```
mapin(ppte, vpagenum, physpagenum, sizeinpages, access)
    struct pte *ppte;
    u_int vpagenum, physpagenum;
    int sizeinpages, access;
```

`mapin()` maps physical addresses to virtual addresses. Device drivers use it to set up kernel virtual memory so that device registers and memory can be directly accessed. This is useful for devices which:

- interface to the kernel by way of two different memory spaces. Since the autoconfiguration process only sets up one space, such cases are best handled by having the *xx*`attach()` routine use `mapin()` to set up the other.
- can consume variable amounts of virtual memory space, and for which, therefore, an optimum mapping cannot be made at autoconfiguration time. This is the case, for example, with certain kinds of variable-resolution frame buffers.

Drivers that call `mapin()` in their *xx*`attach()` routines must first call `rmalloc(kernelmap, ...)` to get the kernel virtual addresses which `mapin()` requires. (Actually, `rmalloc()` will return indexes to kernel virtual addresses—see below). Note that, when a driver calls `mapin()`, it should also call `mapout()` to return the mapped virtual memory when its no longer needed.

ppte is a pointer to the PTE which performs the mapping. This is the PTE in `Sysmap` (defined in `<sun[234]/pte.h>`) which corresponds to the map index returned from `rmalloc(kernelmap, ...)`. That is, *ppte* can be given as `&Sysmap[kmx]`, where kmx is the map index returned by `rmalloc()`.

vpagenum is the number of the virtual page where the physical memory is to be mapped. kmx, the map index returned by `rmalloc()`, can be used to calculate a virtual address, which can then be converted to a virtual page number like so:

```
vpagenum = btoc((Sysbase)) + kmx;
```

Here `Sysmap` is the external array of page table entries used to map virtual addresses, starting at the (kernel virtual) base address `Sysbase`. `btoc()` is a macro (see `machine/param.h`) which converts addresses to page numbers, and, if necessary, performs the appropriate rounding.

Note that there are a number of general-purpose macros designed to convert between kernel map indexes and virtual addresses. These macros are in `<sys/vmmac.h>`. One of them, `kmxtob` expects an (integer) kernel map index and returns the virtual address by page number. Another, `btokmx` expects a (caddr_t) virtual address and returns the integer kernel map index.

physpagenum is the physical page number of the memory being mapped into kernel virtual memory. Actually, it is the physical page number with the appropriate type bits for the given physical memory space—these types bits (PGT_*) are given in `<sys/pte.h>`.

sizeinpages is the size in pages of the memory being mapped. It can be easily computed by using the `btoc()` macro to convert the size (in bytes) of the memory being mapped into pages (since `btoc()` will round up as needed).

access is the PTE-level access flags. The flags (PG_*) are defined in `<sys/pte.h>`. The value passed by the auto-configuration process when it calls `mapin()` (the standard device driver case) is “PG_V|PG_KW”, which indicates valid system pages with their write-enable flags set.

See `fmmapin()` and `fbmapout()` in `fbutils.c` (in the *Sample Driver Listings* appendix) for examples of real `mapin()` and `mapout()` calls.

`mapout()` — Remove Physical to Virtual Mappings

```
mapout (ppte, sizeinpages)
    struct pte *ppte;
    int sizeinpages;
```

`mapout()` is used to unmap a chunk of physical memory from the virtual memory that `mapin()` associated it with. Its parameters are as given in `mapin()`, above. Drivers typically need to call `mapout()` only when they have made their own calls to `rmalloc()` and `rmfree()`. It should be called just before `rmfree()`.

`mbrelse()` — Free Main Bus Resources

```
mbrelse(mb_hd, mbinfop)
    struct mb_hd *mb_hd;
    int *mbinfop;
```

`mbrelse` releases the Main Bus DVMA resources allocated by `mbsetup`. Note that the second parameter is a *pointer* to the integer returned by `mbsetup`.

`mbsetup()` — Set Up to Use Main Bus Resources

```
mbsetup(mb_hd, bp, flag)
    struct mb_hd *mb_hd;
    struct buf  *bp;
    int flag;
```

`mbsetup` is called to set up the memory map for a single Main Bus DVMA transfer. It assumes that *bp*'s fields have been set up to define the transfer, which is generally true, since `physio()` sets them up before calling the driver *xx*`strategy()` routine. (These fields are `b_un.b_addr`, `b_flags` and `b_bcount`). *flag* is `MB_CANTWAIT` if the caller desires not to wait for map resources (slots in the map or DVMA space) if none are available — it's highly unlikely that this will ever happen, but if it does `mbsetup` will return immediately with a 0. In this case its caller can, presumably, wait before trying again. If, on the other hand, *flag* is 0, the requesting process will be put to sleep until the necessary map resources become available.

`mbsetup()` is typically called from the driver `strategy()` routine, so when `physio()` breaks up a large I/O request, one result is the generation of a series of calls to `mbsetup()`. (`mbrelse()` is then called from the driver *xx*`intr()` routine). `mbsetup()`, like `physio()`, is intended primarily for the use of block drivers, though character drivers can use it as long as they don't use buffer headers from the kernel cache. The buffer is *double mapped* so that the system will consider it as being in kernel DVMA space as well as in the address space of the program being serviced.

NOTE *Don't set* `B_PHYS` *in bp's* `b_flags` *field if DVMA is from kernel address space to the device.*

Upon success, `mbsetup` returns an number which must be saved for the call to `mbrelse`. This number can also be passed to `MBI_ADDR()`, which will transform it into a transfer address.

`outb()` — Send a Byte to an I/O Port

```
outb(port, data)
    short port;
    u_char data;
```

Sun386i only. On the Sun386i, many devices, such as the floppy, are accessed by way of the I/O space. `outb()` sends a byte value to the I/O address specified. I/O device addresses are in the range of `0` to `0xFFFF`. (See `inb`()).

`panic()` — Reboot at Fatal Error

```
panic(message)
    char *message;
```

`panic` can be called upon encountering an unresolvable fatal error. It prints its *message* to the system console, and then reboots the system, so don't take its use lightly. (It does have the sense to avoid the reboot if it has already been called — thus preventing recursive calls to `panic`()). A kernel core image is dumped.

`peek()`, `peekc()`, `peekl()` — Check and Read

```
peek(value)
    short *value;

peekc(value)
    char *value;

peekl(address, value)
    long *address;
    long *value;
```

`peek` and its variants are called with an address from which they read. They return −1 if the addressed location doesn't exist, otherwise they return the value that was fetched from that location. They are for use only in *xx*`probe()`. See `poke` and its variants, below.

`physio()` — Block I/O Service Routine

```
physio(strategy, buf, dev, flag, minphys, uio)
    void  (*strategy) ();
    struct  buf  *buf;
    dev_t  dev;
    int  rw_flag;
    void  (*minphys) ();
    struct  uio  *uio;
```

Character drivers sometimes do block I/O, and when they do it's convenient for them to use `physio()`. Such drivers resemble simple block drivers in that they have *xx*`read()` and/or *xx*`write()` and *xx*`strategy()` routines, call those *xx*`strategy()` routines indirectly through `physio()`, and use `buf` structures. Too much, however, should not be made of the similarity. Character-driver *xx*`strategy()` routines typically implement no strategy, and they are not driver entry points. And while character drivers can use `physio()` (and `mbsetup()` and `iowait()` and the few other kernel support routines that manipulate buffer headers) they do not use buffers from the kernel buffer cache.

`physio()` serves two major purposes:

- It ensures that pages of user memory are locked down (physically available and not paged out) during the duration of a data transfer. *This is the only way to lock down pages of user memory.*
- It breaks large transfers (those greater than the value returned by `minphys`()) into smaller pieces, thus keeping slow devices from monopolizing the bus.

If the size of the transfer is greater than the system determined maximum, `physio()` calls the driver *xx*`strategy()` routine repeatedly, making sure that all relevant pointers and counters are updated correctly. Basically, `physio()` looks like:

```
loop:
    error and termination checking (based on values in uio)
    s = spl6();
    while (buf->b_flags & B_BUSY) {
        buf->b_flags |= B_WANTED;
        sleep(buf);
    }
    (void) splx();
    set up buffer for I/O;
    while (more data) {
        buf->b_flags = B_BUSY | B_PHYS | rw_flag;
        more buffer I/O set up;
        (*minphys) (buf);
        lock down pages of user memory
        (*strategy) ();
        spl6();
        unlock buffer;
        if (buf->b_flags & B_WANTED)
            wakeup(buf);
        (void) splx(s);
        bookkeeping;
    }
    buf->b_flags &= ~(B_BUSY|B_WANTED|B_PHYS);
    error checking and bookkeeping (based on values in uio)
    goto loop:
```

`buf` is a buffer header for this device. `physio()` wants exclusive use of this buffer header and its associated buffer, and when called it checks to see if it has it. If it doesn't, it will `sleep()` until it gets it. `dev` is the device to which the transfer is taking place. *rw_flag* is `B_READ` or `B_WRITE` to indicate the direction of the transfer. `minphys()` is a function that determines the amount of data to be transferred in one call to the *xx*`strategy()` routine. *uio* is a pointer to the `uio` structure.

`physio()` returns one of the error codes defined in `errno.h` if an I/O error occurs, and a 0 upon success. Error codes are not returned on the stack, but by way of the `b_error` field in the buffer header.

poke(), **pokec()**, **pokel()** — Check and Write

```
poke(address, value)
    short *address;
    short value;

pokec(address, value)
    char *address;
    char value;

pokel(address, value)
    long *address;
    long value;
```

`poke` and its variants are called with an *address* to store into, and a *value* to be stored. They return 1 if the addressed location doesn't exist, and 0 if it does. They are for use only in *xx*`probe()`. See `peek` and its variants, above.

`printf()` — Kernel Printf Function

The kernel provides a `printf()` function analogous to the `printf()` function supplied with the C library for user programs. The kernel `printf()`, however, is more limited than is the version in the C library. It writes directly to the console tty, its output cannot be easily redirected, and it supports only a subset of `printf()`'s formatting conversions. Furthermore, it's not interrupt driven, and thus causes all system activities to be suspended while it outputs its message. Nevertheless, `printf()` is useful as a debugging tool, and for reporting error messages. See `uprintf()`.

The formatting conversions supported by the kernel `printf()` are:

`%x, %X` - Hexadecimal numbers
`%d, %D` - Decimal numbers
`%o, %O` - Octal numbers
`%c` - Single characters
`%s` - Strings
`%b` - Bit values

Note that floating-point conversions are *not* supported. Also note that a special format %b is provided to decode error registers. Its usage is:

```
printf("reg=%b\n", regval, "<base><arg>*");
```

Where `<base>` is the output base expressed as a control character. For example, `\10` gives octal and `\20` gives hex. Each `arg` is a sequence of characters, the first of which gives the bit number to be inspected (counting from 1), and the rest of which (up to a control character, that is, a character <= 32), give the name of the register. Thus:

```
printf("reg=%b\n", 3, "\10\2BITTWO\1BITONE\n");
```

would produce the output:

```
reg=3<BITTWO,BITONE>
```

Also note that no conversion modifiers (field widths and so on) are supported — only a single character can follow the `%`.

The kernel `printf()` function raises the priority level and therefore locks out interrupts while it is sending data to the console. And it displays its messages directly on the console, unless specifically redirected by the `TIOCCONS` ioctl.

`pritospl()` — Convert Priority Level

```
pritospl(value)
    int  value;
```

`pritospl` is a macro that converts the hardware priority level given by *value*, which is a Main Bus priority level, to the processor priority level that `splx` expects. The Main Bus priority level can be found in either `mb_device.md_intpri` or `mb_ctlr.mc_intpri`, where it is put by the autoconfiguration process. `pritospl` is used to parameterize the setting of priority levels. See `spl`*n* and `splx()`.

`psignal()` — Send Signal to Process

```
psignal(p, sig)
    struct proc *p;
    int sig;
```

Sends signal *sig* to the process specified by the `proc` structure. See `gsignal()`.

`rmalloc()` — General-Purpose Resource Allocator

```
u_long rmalloc(mp, size)
    struct map *mp;
     long size;
```

`rmalloc` (for resource map allocator) is a rather specialized sort of resource allocator. In fact, it doesn't really allocate resources at all, but rather names of resources (that is, lists of numbers). Such lists are initialized by `rminit()` and are called resource "maps". Given such a map, `rmalloc()` can parcel out the names in it. The relationship of such names to real resources (virtual address space, physical memory, and so on) is entirely a matter of usage conventions. Names allocated with `rmalloc()` are recycled with `rmfree`.

`rmalloc` is a low-level routine, and shouldn't be used casually. If you just want some kernel virtual memory, use `kmem_alloc()`. `rmalloc()` is called by drivers that need to allocate kernel virtual address space during their *xx*`probe()` and *xx*`attach()` routines. They call it, rather than `kmem_alloc()`, because they want an address space without physical memory mapped to it.

`rminit()` is *not* documented here, for device drivers only have occasion to use two pre-initialized `rmalloc()` maps:

- The map `kernelmap` (in `<sys/map.h>`) is used to allocate chunks of generic kernel virtual address space.
- The map `iopbmap` (in `<sundev/mbvar.h>`) contains addresses that are guaranteed to be in the high megabyte and thus suitable for use as DVMA buffer addresses. `iopbmap` is quite small, and should be used only for temporary or very small buffers.

`rmfree()` — Recycle Map Resource

```
rmfree(mp, size, addr)
    struct map *mp;
    long size;
    u_long addr;
```

`rmfree` recycles the map resource allocated with `rmalloc`.

`selwakeup()` — Wakeup a Select-blocked Process

```
selwakeup(p, coll)
    register struct proc *p;
    int coll;
```

`selwakeup()` is called from driver interrupt routines to `wakeup()` processes which are asleep as a result of calls to `select()`. If both of its parameters are 0, it does nothing. If *coll* is 0, thus indicating that no `select()` collision occurred — that only one process is waiting for the device — `selwakeup()` just wakes up the waiting process indicated by *p*. If, however, a collision did occur, it issues a `wakeup((caddr_t)&selwait)`, thus waking all select-sleeping processes. (The `selwait` channel is used exclusively to indicate select-related sleeping). These waking processes then race for access to the device, with the first selector getting no special treatment.

`sleep()` — Sleep on an Event

```
sleep(address, priority)
    caddr_t address;
    int priority;
```

`sleep` is called to put the calling process to sleep, typically while it awaits the availability of some system resource. *address* is the address of a location in memory, usually a field in some global driver structure that is being used as a "semaphore" (such fields are not true semaphores, see below). *priority* is the software priority the calling process will have after being awakened.

`sleep` must *never* be called from the interrupt-level side of a driver. This is because `sleep()` is always executed on behalf of a specific process. It suspends that process while the scheduler picks and executes another waiting process. And since, when handling an interrupt, the kernel isn't running on behalf of any process, it makes no sense to call `sleep()`. Incidently, the kernel will `panic()` if sleep is called while it's running on the interrupt stack.

A process that has called `sleep()` will be reawakened by any `wakeup` call issued with the same *address*. However *it's not guaranteed that, upon waking, the process will find the resource that it was waiting for to be available.* It must, therefore, check again before proceeding, and go back to sleep if necessary. This is because the SunOS `sleep()` and `wakeup()` facilities do not constitute true semaphore primitives in the usual P/V sense. `wakeup` will wakeup *every process* that is sleeping on that event, where a true 'V' semaphore will wake only one sleeper (the highest priority one or whichever).

Thus in SunOS you always do:

```
s = spln(high_priority);
while (resource_busy)
    sleep(resource, high_priority);
make_resource_busy;
(void) splx(s);
. . .
<critical section>
. . .
wakeup(resource);
```

whereas with real semaphores you would simply do:

```
P(resource);
. . .
<critical section>
. . .
V(resource);
```

which is a much simpler and cleaner design.

However, semaphores are not easy to use to implement lockouts around hardware interrupts so SunOS just uses the `sleep()/wakeup()` mechanism for both situations.

spl*n*() — Set CPU Priority Level

The `spl`*n* functions are available for setting the CPU priority level to *n*, where *n* ranges from 0 to 7 (higher numbers indicate higher priorities). Note that `spl6()` actually gets you `spl5()` on Sun systems to avoid lockout of the level 6 on-board UART interrupts. When you allocate a CPU priority level to your device, choose one that's high enough to give you the performance you need, but don't overdo it or you will interfere with the operation of the system:

- If you lock out the on-board UARTS (level 6) characters may be lost.
- If you lock out the clock (level 5) time will not be accurate, and the SunOS scheduler will be suspended.
- If you lock out the Ethernet (level 3), packets may be lost and retransmissions needed.
- And if you lock out the disks (level 2), disk rotations may be missed.

The `spl`*n* functions return the previous priority level.

splx() — Reset Priority Level

```
splx(s)
   int  s;
```

`splx` called with an argument *s* sets the priority level to *s*, which was returned from a previous call to `spl`*n*, `pritospl()`, or `splx()`. `splx` is typically used to restore the priority level to a previously stored level. `splx()` returns the previous level.

`suser()` — Reset Priority Level

```
suser()
```

Returns a 1 if the current user is root, 0 if not. `suser()` is commonly called by `ioctl()` routines that are restricted to the superuser, and that thus need to check who's calling them.

`swab()` — Swap Bytes

```
swab(from, to, nbytes)
    caddr_t  from;
    caddr_t  to;
    int  nbytes;
```

`swab` swaps bytes within 16-bit words. *nbytes* is the number of bytes to swap, and is rounded up to a multiple of two. No checking is done to ensure that the *from* and *to* areas do not overlap each other.

`timeout()` — Wait for an Interval

```
timeout(func, arg, interval)
    int (*func)();
    caddr_t arg;
    int interval;
```

`timeout` arranges that after *interval* clock-ticks, *func* will be called with *arg* as its argument, in the style *(*func)(arg)*. A clock tick is about a fiftieth of a second for Sun-2, Sun-3, and Sun386i machines, a hundredth of a second for Sun-4s. The precise number of clock ticks per second is given in the external variable `hz`. Timeouts are used, for example, to provide real-time delays after function characters like new-line and tab in typewriter output, and to cancel read or write requests that have received no response within a specified amount of time (if there's a lost interrupt or if the device otherwise flakes out). The specified *func* is eventually called from the lower half of the clock-interrupt routine, so it must conform to the requirements of interrupt routines in general. In particular, it can't call `sleep()`. See `untimeout()`.

`uiomove()` — Move Data To or From an **`uio`** Structure

```
uiomove(cp, n, rw, uio)
    caddr_t  cp;
    int n;
    enum uio_rw  rw;
    struct *uio;
```

`uiomove()` is the most common way for device drivers to move a specified number of bytes between a byte array in kernel address space and an area defined by a `uio` structure (which may or may not be in kernel address space). If the `uio_seg` field in the `uio` structure is set to `UIOSEG_USER`, `uiomove()` will assume the *uio* pointer to be in user space; if it is `UIOSEG_KERNEL`, it will assume it to be in kernel space (see `<sys/uio.h>`). `uiomove()` moves *n* bytes between the `uio` structure and the area defined by the *cp* parameter. The read/write flag is interpreted as follows: — `UIO_READ` indicates a transfer from kernel to user space (a call to `copyout`()), and `UIO_WRITE` a transfer from user to kernel space (a call to `copyin`()). `uiomove()` returns 0 upon success, `Exxx` upon failure.

For more information about the `uio` structure, see *Some Notes About the UIO Structure* in the *The "Skeleton" Character Device Driver* chapter of this manual.

untimeout() — Cancel **timeout** Request

```
untimeout(func, arg)
    int (*func)();
    caddr_t arg;
```

`untimeout` is called to cancel a prior `timeout` request. *func* and *arg* are the same as in `timeout()`.

uprintf() — Nonsleeping Kernel Printf Function

`uprintf()` is like `printf()`, with two important differences. The first is that it checks to see if the process' "controlling terminal" is open, and if it is the message is sent to it rather than to the system console (`uprintf()` consults the `user` structure, so it must not be called from the lower-half routines). If there's no controlling terminal, `uprintf()` executes as would `printf()`. The second difference is that `uprintf()` is interruptible, and thus reasonably efficient.

`uprintf()` is often called from `open()` routines to report errors to the user. It's used for errors which, like tape-read errors, are likely to indicate operator error rather than system failure. See `printf()`.

ureadc(), **uwritec()** — `uio` Structure Read/Write

```
ureadc(c, uio)
    int c;
    struct *uio;
```

`ureadc()` transfers the character *c* into the `uio` structure (which is normally passed to the driver when it is called). `ureadc()` is normally used when "reading" a character in from a device.

```
uwritec(uio)
    struct *uio;
```

`uwritec()` returns the next character in the `uio` structure (which is normally passed to the driver when it is called), or returns –1 on error. `uwritec()` is normally used when "writing" a character to a device.

Note that "read" and "write" are slightly confusing in the above contexts, since `ureadc()` actually obtains a character from somewhere and places it *into* the `uio` structure, whereas `uwritec()` obtains a character from the `uio` structure and "writes" it somewhere else. The "read" and the "write," then, are from the perspective of the user program.

`ureadc()` and `uwritec()` replace the routines `cpass()` and `passc()`, which are no longer supported.

wakeup() — Wake Up a Process Sleeping on an Event

```
wakeup(address)
    caddr_t address;
```

`wakeup` is called when a process waiting on an event must be awakened. *address* is typically the address of a location in memory. `wakeup` is typically called from the low level side of a driver when (for instance) all data has been transferred to or from the user's buffer and the process waiting for the transfer to complete must be awakened. See `sleep()`.

D

User Support Routines

User Support Routines **385**

`free()` — Free Allocated Memory 385
`getpagesize()` — Return Pagesize 385
`mmap()` — Map Memory from One Space to Another 385
`munmap()` — Unmap Pages of Memory 386

D

User Support Routines

These routines are often useful in user-level programs that manipulate devices.

`free()` — Free Allocated Memory

```
free(ptr)
    char *ptr;
```

`free(3)` can be used to recycle the virtual memory allocated by a variety of memory allocators, including `valloc(3)` and `malloc(3)` (the most general purpose of the allocators).

`getpagesize()` — Return Pagesize

```
int getpagesize()
```

`getpagesize(2)` returns the number of bytes in a page. The page size is the system page size and may not be identical with the page size in the underlying hardware — it is, however, the pagesize of interest in all of the memory management functions.

`mmap()` — Map Memory from One Space to Another

```
caddr_t
mmap(addr, len, protection, flags, fd, off)
    caddr_t addr;
    int len, protection, flags, fd;
    off_t off;
```

`mmap()` maps pages of memory space from the memory device associated with the file *fd* into the address space of the calling process (or into the kernel address space). The mapping is performed one page at a time, by iteratively calling the memory device's `mmap()` routine.

The memory is mapped from the memory device, beginning at *off* (the device's physical installation address within *fd*'s memory), into the caller's address space beginning at *addr* and continuing for *len* bytes. (By default, `mmap()` will pick a good value for *addr*). The mapping established by `mmap()` replaces any previous mappings for the process's pages in the range [*addr*, *addr* + *len*).

fd is a file descriptor obtained by opening the character special device to be `mmap()`'ed. *protection* specifies the read/write accessibility of the mapped pages. The values desired are expressed by or'ing the flags values `PROT_READ`, `PROT_EXECUTE`, and `PROT_WRITE`. A `write()` must fail if `PROT_WRITE` has not been set, though its behavior can be influenced by setting `MAP_PRIVATE` in the *flags* parameter.

flags provides additional information about the handling of mapped pages. Its possible values are:

`MAP_SHARED`	Share Changes
`MAP_PRIVATE`	Changes are Private
`MAP_TYPE`	Mask for Type of Mapping
`MAP_FIXED`	Interpret addr Exactly
`MAP_RENAME`	Assign Page to File

addr and *off* must be multiples of the page size (which can be found by using `getpagesize()`). Pages are automatically unmapped when *fd* is closed — they should be explicitly unmapped with `munmap()`. `mmap()` returns a -1 on error, 0 on success.

For an detailed overview of SunOS memory mapping, see the *Memory Management* chapter of the *Sun System Services Overview.* For specific details about `mmap()` and its related facilities, see `munmap()` below and the `mmap(2)`, `munmap(2)`, `mincore(2)`, `mprotect(2)`, and `msync(2)` manual pages.

munmap() — Unmap Pages of Memory

```
munmap(addr, len)
    caddr_t addr;
    int len;
```

`munmap()` causes the pages starting at *addr* and continuing for *len* bytes to be unmapped, that is, marked invalid. If an address within an unmapped page is subsequently referenced, and if that page is in the "data segment" of a UNIX process, then a page of zeros will be created under the address. However, if the address is outside a data segment, such a reference will cause a segmentation violation. `munmap()` returns a -1 on error, 0 on success. See `mmap()` above and the `mmap(2)` manual page for more details.

E

Sample Driver Listings

Sample Driver Listings **389**

E.1. Skeleton Board Driver 390
E.2. Sun-2 Color Graphics Driver 398
E.3. Sky Floating-Point Driver 415
E.4. Versatec Interface Driver 423
E.5. Sun386i Parallel Port Driver 435

E

Sample Driver Listings

The following source listings are for sample Sun device drivers. There are four drivers listed here; the first being the skeleton driver and the other three being real production drivers. (These three drivers, it should be mentioned, have been chosen as relatively simple illustrations of the three major types of drivers — not as software ideals to be closely emulated).

SKELETON
is the driver for the "skeleton board" discussed earlier in this manual.

CGTWO
is a device driver for the Sun-2 Color Graphics board. It is one of the simplest drivers around, being memory mapped.

SKY
is a programmed I/O driver for the Sky floating-point board, with both polling interrupts and vectored interrupts. However, the interrupt routines don't do a whole lot.

VP is a driver for the Versatec Printer Interface. It's a fairly good example of a DMA device driver.

PP is the listing of the Sun386i Parallel Port Driver.

E.1. Skeleton Board Driver

```
/*
 * (skreg.h) Registers for Skeleton Board -- note the byte swap
 */

struct sk_reg {
     char sk_data;    /* 01: Data Register */
     char sk_csr;     /* 00: command(w) and status(r) */
};

/* sk_csr bits (read) */
#define    SK_INTR          0x80  /* Device is Interrupting*/
#define    SK_DEVREADY      0x08  /* Device is Ready */
#define    SK_INTREADY      0x04  /* Interface is Ready */
#define    SK_ERROR         0x02  /*  Device Error  */
#define    SK_INTENAB       0x01  /* Interrupts are Enabled */

#define    SK_ISTHERE       0x0C  /*  Existance Check; Device and Interface Ready  */

/* sk_csr bits (write) */
#define    SK_RESET   0x04        /* Reset Device and Interface */
#define    SK_ENABLE  0x01        /* Enable Interrupts */


/*
 * Further definitions for DMA skeleton board
 */

#define    SK_DMA     0x10        /* Do DMA transfer */
#define    MAX_SK_BSIZE 4096      /* DMA tranfer block */

struct sk_reg2 {
     char sk_data;          /* 01: Data Register */
     char sk_csr;           /* 00: command(w) and status(r) */
     short sk_count;        /* bytes to be transferred */
     caddr_t sk_addr;       /* DMA address */
};
```

```
/*
 * (sk.c) The "Skeleton Board" Driver"
 */

/* This listing is not heavily annotated. This is because it's identical to
 * the Skeleton driver discussed at length in the main body of the manual.
 * It appears here for purposes of completeness.
 */

#include <sys/param.h>
#include <sys/buf.h>
#include <sys/file.h>
#include <sys/dir.h>
#include <sys/user.h>
#include <sys/uio.h>
#include <machine/psl.h>
#include <sundev/mbvar.h>

#include "sk.h"        /* file generated by config (defines NSK) */
#include "skreg.h"     /* register definitions */

#define SKPRI (PZERO-1)   /* software sleep priority for sk */

#define SKUNIT(dev) (minor(dev))

struct buf skbufs[NSK];

int skprobe(), skpoll();

struct mb_device *skdinfo[NSK];
struct mb_driver skdriver = { skprobe, 0, 0, 0, 0, skpoll,
     sizeof(struct sk_reg), "sk", skdinfo, 0, 0, 0, 0,
};

struct sk_device {
     char soft_csr;          /* software copy of control/status register */
     struct buf *sk_bp;      /* current buf */
     int sk_count;           /* number of bytes to send */
     char *sk_cp;            /* next byte to send */
     char sk_busy;           /* true if device is busy */
} skdevice[NSK];

/*ARGSUSED*/
skprobe(reg, unit)
     caddr_t reg;
     int unit;
{
     register struct sk_reg *sk_reg;
     register int c;

     sk_reg = (struct sk_reg *)reg;

     c = peekc((char *)&sk_reg->sk_csr);   /* contact the device */
```

```
        if (c == -1 || (c != SK_ISTHERE))
                return (0);
        if (pokec((char *)&sk_reg->sk_csr, SK_RESET)) /* contact the device */
                return (0);

        return (sizeof (struct sk_reg));
}

skopen(dev, flags)
        dev_t dev;
        int flags;
{
        register int unit = SKUNIT(dev);
        register struct mb_device *md;
        register struct sk_reg *sk_reg;

        md = skdinfo[unit];

        if (unit >= NSK || md->md_alive == 0)
                return (ENXIO);
        if (flags & FREAD)
                return (ENODEV);

        sk_reg = (struct sk_reg *)md->md_addr;

        /* enable interrupts */
        skdevice[unit].soft_csr = SK_ENABLE;

        /* contact the device */
        sk_reg->sk_csr = skdevice[unit].soft_csr;

        return (0);
}

/*ARGSUSED*/
skclose(dev, flags)
        dev_t dev;
        int`flags;
{
        register int unit = SKUNIT(dev);
        register struct mb_device *md;
        register struct sk_reg *sk_reg;

        md = skdinfo[unit];

        /* disable interrupts */
        sk_reg = (struct sk_reg *)md->md_addr;
        skdevice[unit].soft_csr &= ~SK_ENABLE;

        /* contact device */
        sk_reg->sk_csr = skdevice[unit].soft_csr;
}
```

```
skminphys(bp)
     struct buf *bp;
{

     if (bp->b_bcount > MAX_SK_BSIZE)
          bp->b_bcount = MAX_SK_BSIZE;
}

skstrategy(bp)
     register struct buf *bp;
{
     register struct mb_device *md;
     register struct sk_device *sk;
     int s;

     md = skdinfo[SKUNIT(bp->b_dev)];     /* physio put the device number into bp */
     sk = &skdevice[SKUNIT(bp->b_dev)];

     s = splx(pritospl(md->md_intpri)); /* begin critical section */
     while (sk->sk_busy)
          sleep((caddr_t) sk, SKPRI);

     /* set up for first write */
     sk->sk_busy = 1;
     sk->sk_bp = bp;
     sk->sk_cp = bp->b_un.b_addr;
     sk->sk_count = bp->b_bcount;
     skstart(sk, (struct sk_reg *)md->md_addr);

     (void) splx(s);      /* end critical section */
}

skwrite(dev, uio)
     dev_t dev;
     struct uio *uio;
{
     register int unit = SKUNIT(dev);

     if (unit >= NSK)
          return (ENXIO);
     return (physio(skstrategy, &skbufs[unit],
          dev, B_WRITE, skminphys, uio));
}

skstart(sk, sk_reg)
     struct sk_device *sk;
     struct sk_reg *sk_reg;
{

     while(sk->sk_count > 0) {   /* still more characters */
          sk_reg->sk_data = *sk->sk_cp++;
          sk->sk_count--;
          /* stop giving characters if device not ready */
          /* Note: the softcopy isn't needed for reads */

          /* DELAY(10) might go here */

          if (!(sk_reg->sk_csr & SK_DEVREADY))   /* contact the device */
               break;
     }

     /* error-retry logic would go here */

     if (sk->sk_count > 0) {   /* still more characters */
          sk->soft_csr = SK_ENABLE;
          sk_reg->sk_csr = sk->soft_csr; /* contact the device */
     } else {
          /* special case: finished the command without taking any interrupts! */
          sk->soft_csr = 0;    /* disable interrupts */
          sk_reg->sk_csr = sk->soft_csr;      /* contact the device */
          sk->sk_busy = 0;
          wakeup((caddr_t) sk);      /*free device to sleeping strategy routine */
          iodone(sk->sk_bp);   /*free buffer to waiting physio */
     }
}

skpoll()
{
     register struct sk_reg *sk_reg;
     int serviced, i;

     serviced = 0;
     for (i = 0; i < NSK; i++) {    /* try each one */
          sk_reg = (struct sk_reg *)skdinfo[i]->md_addr;
          if (sk_reg->sk_csr & SK_INTR) { /* contact the device */
               serviced = 1;
               skintr(i);
          }
     }
     return (serviced);
}

skintr(i)
     int  i;
{
     register struct sk_reg *sk_reg;
     register struct sk_device *sk;

     sk_reg = (struct sk_reg *)skdinfo[i]->md_addr;
     sk = &skdevice[i];

     /* check for an I/O error */
     if (sk_reg->sk_csr & SK_ERROR) { /* contact the device */

          /* error-retry logic would go here */
          printf("skintr: I/O error\n");
          sk->sk_bp->b_flags |= B_ERROR;
          goto error_return;
     }

     if (sk->sk_count == 0) { /* I/O transfer completed */
error_return:
          sk->soft_csr = 0;  /* clear interrupt */
          sk_reg->sk_csr = sk->soft_csr; /* contact the device */
          sk->sk_busy = 0;
          wakeup((caddr_t) sk); /* free device to sleeping strategy routine */
          iodone(sk->sk_bp); /* free buffer to waiting physio */
     } else skstart(sk, sk_reg);
}


/* DMA VARIATIONS FOLLOW */

struct sk_device {
     char soft_csr;           /* software copy of control/status register */
     struct buf *sk_bp;       /* current buf */
     char sk_busy;            /* true if device is busy */
     int sk_mbinfo;           /* Information stash for DMA */
} skdevice[NSK];

skstrategy(bp)
     register struct buf *bp;
{
     register struct mb_device *md;
     register struct sk_reg *sk_reg;
     register struct sk_device *sk;
     int s;

     md = skdinfo[SKUNIT(bp->b_dev)];
     sk_reg = (struct sk_reg *)md->md_addr;
     sk = &skdevice[SKUNIT(bp->b_dev)];

     s = splx(pritospl(md->md_intpri));      /* begin critical section */
     while (sk->sk_busy)
          sleep((caddr_t) sk, SKPRI);
     sk->sk_busy = 1;
     sk->sk_bp = bp;

     /* this is the part that is changed */

     /* grab bus resources */
     sk->sk_mbinfo = mbsetup(md->md_hd, bp, 0);

     /* the remainder */
     sk_reg->sk_count = bp->b_bcount;

     /* plug bus transfer address */
     sk_reg->sk_addr = (caddr_t)MBI_ADDR(sk->sk_mbinfo);
```

```
        /* make sure we didn't overrun the address space limit */
        if (sk_reg->sk_addr > (caddr_t) 0x000FFFFF) {
                printf("sk%d: ", sk_reg->sk_addr);
                panic("exceeded 20 bit address");
        }

        sk->soft_csr = SK_ENABLE | SK_DMA;
        sk_reg->sk_csr = sk->soft_csr;      /* contact the device */

        /* end of DMA-related changes */

        (void) splx(s);       /* end critical section */
}

skpoll()
{
        register struct mb_device *md;
        register struct sk_reg *sk_reg;
        int serviced, i;

        serviced = 0;
        for (i = 0; i < NSK; i++) {
                md = (struct mb_device *)skdinfo[i];
                sk_reg = (struct sk_reg *)md->md_addr;
                if (sk_reg->sk_csr & SK_INTR) {
                        serviced = 1;
                        skintr(i);
                }
        }
        return (serviced);
}

skintr(i)
        int  i;
{
        register struct mb_device *md;
        register struct sk_reg *sk_reg;
        register struct sk_device *sk;

        md = (struct mb_device *)skdinfo[i];
        sk_reg = (struct sk_reg *)md->md_addr;
        sk = &skdevice[i];

        /* check for an I/O error */
        if (sk_reg->sk_csr & SK_ERROR) { /* contact the device */

                /* error-retry logic would go here */

                printf("skintr: I/O error0);
                sk->sk_bp->b_flags |= B_ERROR;
        }

        /* this is the part that changed */
        sk->soft_csr = 0;    /* clear interrupt */
        sk_reg->sk_csr = sk->soft_csr;
        mbrelse(md->md_hd, &sk->sk_mbinfo);
        sk->sk_busy = 0;
        wakeup((caddr_t) sk); /* free device to sleeping strategy routine */
        iodone(sk->sk_bp);   /* free buffer to waiting physio */
}
```

E.2. Sun-2 Color Graphics Driver

```
/*
 *
 * (cg2reg.h) Description of SUN-2 hardware color frame buffer.
 * Copyright (c) 1983 by Sun Microsystems, Inc.
 */

/*
 * Structure defining the way in which the address bits to the
 * SUN-2 color frame buffer are decoded.
 */

#define CG2_WIDTH   1152
#define CG2_HEIGHT  900
#define CG2_SQUARE  1024
#define CG2_DEPTH   8

struct cg2memfb {
     union bitplane {              /* Word mode memory */
          short word[CG2_HEIGHT][CG2_WIDTH/(8*sizeof(short))];
          short sword[CG2_SQUARE][CG2_SQUARE/(8*sizeof(short))];
     } memplane[8];
     union byteplane {             /* Pixel mode memory */
          u_char pixel[CG2_HEIGHT][CG2_WIDTH];
          u_char spixel[CG2_SQUARE][CG2_SQUARE];
     } pixplane;
};

struct cg2statusreg {
     unsigned unused    : 4;       /* Reserved for future use */
     unsigned resolution : 4;      /* Screen resolution */
                                        /* 0 = 900 x 1152 */
                                        /* 1 = 1024 x 1024 */
     unsigned retrace : 1;         /* rdonly: monitor in retrace */
     unsigned inpend  : 1;         /* rdonly: interrupt pending */
     unsigned ropmode : 3;         /* Rasterop mode */
     unsigned inten   : 1;         /* Enable interrupt at end of retrace */
     unsigned update_cmap : 1;
                                   /* Copy TTL cmap to ECL cmap next vert retrace*/
                                   /* Silently disables writing to TTL cmap */
     unsigned video_enab : 1;      /* Enable video DACs */
};

struct cg2fb {
     union {                       /* ROP mode memory */
          union bitplane ropplane[8];   /* Word mode memory with ROP */
          union byteplane roppixel;     /* Pixel mode memory with ROP */
     } ropio;
     union {                       /* Rasterop unit control */
```

```
          struct memropc ropregs;  /* Normal register access */
          struct {
               char pad[2048];     /* For pixmode src reg prime */
               struct memropc ropregs; /* Byte xfer loads alternate */
          } prime;                 /* Source register bits */
          char pad[4096];
     } ropcontrol[9];
     union {                       /* Status register */
          struct cg2statusreg reg;
          short word;
          char pad[4096];
     } status;
     union {                       /* Per plane mask register */
          unsigned short reg;      /* 8 bits 1bit -> wr to plane*/
          char pad[4096];
     } ppmask;
     union {                       /* Word pan register */
          unsigned short reg;      /* High 16 bits of 20-bit pixel address*/
                                   /* Pixel addr = CG2_WIDTH*y+x */
          char pad[4096];
     } wordpan;
     union {                       /* Zoom and line offset register */
          struct {
              unsigned unused  : 8;
              unsigned lineoff : 4;/* y offset into zoomed pixel */
              unsigned pixzoom : 4;/* Zoomed pixel size - 1 */
          } reg;
          short word;
          char pad[4096];
     } zoom;
     union {                       /* Pixel pan register */
          struct {
              unsigned unused   : 8;
              unsigned lorigin  : 4;    /* Low 4 bits of pix addr*/
              unsigned pixeloff : 4;    /* Zoomed pixel x offset/4 */
          } reg;
          short word;
          char pad[4096];
     } pixpan;
     union {                       /* Variable zoom register */
                                   /* Reset zoom after line no */
          unsigned short reg;      /* Line number 0..1024/4  */
          char pad[4096];
     } varzoom;
     union {                       /* Interrupt vector register */
          unsigned short reg;      /* Line number 0..1024/4  */
          char pad[4096];
     } intrptvec;
     u_short redmap[256];          /* Shadow color maps */
     u_short greenmap[256];
     u_short bluemap[256];
};
```

```
/*
 * ROPMODES -- Parallel, LD_SDT, LS_SRC, Read/Write,
 *                 on read or write?, on wrdmode or pixmode?
 */

#define   PRWWRD    0    /* parallel  8 plane,  read   write,   wrdmode */
#define   SRWPIX    1    /* single      pixel,  read   write,   pixmode */
#define   PWWWRD    2    /* parallel  8 plane,  write  write,   wrdmode */
#define   SWWPIX    3    /* single      pixel,  write  write,   pixmode */
#define   PRRWRD    4    /* parallel  8 plane,  read   read,    wrdmode */
#define   PRWPIX    5    /* parallel 16 pixel,  read   write,   pixmode */
#define   PWRWRD    6    /* parallel  8 plane,  write  read,    wrdmode */
#define   PWWPIX    7    /* parallel 16 pixel,  write  write,   pixmode */

/*
 * ROP control unit numbers
 */

#define CG2_ROP0     0    /* Rasterop unit for bit plane 0 */
#define CG2_ROP1     1    /* Rasterop unit for bit plane 1 */
#define CG2_ROP2     2
#define CG2_ROP3     3
#define CG2_ROP4     4
#define CG2_ROP5     5
#define CG2_ROP6     6
#define CG2_ROP7     7
#define CG2_ALLROP   8    /* Writes to all units enabled by PPMASK, */
                          /* reads from plane zero */

#define   CG_SRC          0xCC
#define   CG_DEST         0xAA
#define   CG_MASK         0xf0
#define   CG_NOTMASK      0x0f
#define   CGOP_NEEDS_MASK(op)       ( (((op)>>4)^(op)) & CG_NOTMASK)

/*
 * Defines for accessing the rasterop units
 */

#define   cg2_setrsource(fb, ropunit, val)\
          ((fb)->ropcontrol[(ropunit)].ropregs.mrc_source1 = (val))
#define   cg2_setlsource(fb, ropunit, val)\
          ((fb)->ropcontrol[(ropunit)].ropregs.mrc_source2 = (val))
#define   cg2_setfunction(fb, ropunit, val)\
          ((fb)->ropcontrol[(ropunit)].ropregs.mrc_op = (val))
#define   cg2_setpattern(fb, ropunit, val)\
          ((fb)->ropcontrol[(ropunit)].ropregs.mrc_pattern = (val))
#define   cg2_setshift(fb, ropunit, shft, dir)\
          ((fb)->ropcontrol[(ropunit)].ropregs.mrc_shift =\
           (shft)|((dir)<<8)  )
#define   cg2_setwidth(fb, ropunit, w, count)\
          ((fb)->ropcontrol[(ropunit)].ropregs.mrc_width = (w));\
           ((fb)->ropcontrol[(ropunit)].ropregs.mrc_opcount = (count))
```

```
/*
 * Defines for accessing the zoom and pan registers
 */

#define    cg2_setzoom(fb, pixsize)\
           ((fb)->zoom.reg.pixzoom = (pixsize)-1)
#define    cg2_setpanoffset(fb, xoff, yoff)\
           ((fb)->pixpan.reg.pixeloff = (xoff)>>2;\
            (fb)->zoom.reg.lineoff = (yoff)
#define    cg2_setpanorigin(fb, x, y)\
           ((y) = ((fb)->status.reg.resolution == 1) ?\
            (y)*CG2_SQUARE+(x) : (y)*CG2_WIDTH+(x);\
            (fb)->pixpan.reg.lorigin = (y)&0xf;\
            (fb)->wordpan.reg = (y)>>4)
#define    cg2_setzoomstop(fb, y) ((fb)->varzoom.reg = (y)>>2)

/*
 *   Defines that facilitate addressing the frame buffer
 */

#define    cg2_pixaddr(fb, x, y)\
           (((fb)->status.reg.resolution) ?\
            &(fb)->pixplane.spixel[(y)][(x)] :\
            &(fb)->pixplane.pixel[(y)][(x)] )
#define    cg2_wordaddr(fb, plane, x, y)\
           (((fb)->status.reg.resolution) ?\
            &(fb)->memplane[(plane)].sword[(y)][(x)>>4] :\
            &(fb)->memplane[(plane)].word[(y)][(x)>>4])
#define    cg2_roppixaddr(fb, x, y)\
           (((fb)->status.reg.resolution) ?\
            &(fb)->ropio.roppixel.spixel[(y)][(x)] :\
            &(fb)->ropio.roppixel.pixel[(y)][(x)])
#define    cg2_ropwordaddr(fb, plane, x, y)\
           (((fb)->status.reg.resolution) ?\
            &(fb)->ropio.ropplane[(plane)].sword[(y)][(x)>>4]:\
            &(fb)->ropio.ropplane[(plane)].word[(y)][(x)>>4])
#define    cg2_width(fb )                             \
           ( ((fb)->status.reg.resolution) ? CG2_SQUARE : CG2_WIDTH )
#define    cg2_height(fb )\
           ( ((fb)->status.reg.resolution) ? CG2_SQUARE : CG2_HEIGHT )
#define    cg2_linebytes(fb, mode)\
           ( ((fb)->status.reg.resolution)\
            ? ( ((mode)&1)?CG2_SQUARE:CG2_SQUARE/8 )\
            : ( ((mode)&1)?CG2_WIDTH:CG2_WIDTH/8   ))
#define    cg2_prskew(x) ((x) & 15)
#define    cg2_touch(a) ((a)=0)
```

```
/* (cg2var.h) More Sun-2 color frame buffer definitions
 * Copyright (c) 1983 by Sun Microsystems, Inc.
 */

/*
 * Information pertaining to the Sun-2 color buffer but not to pixrects in
 * general is stored in the struct pointed to by the pr_data attribute of the
 * pixrect. One property of the color buffer not shared with all pixrects is
 * that it has a color map. The color map type and colormap contents are
 * specified by the putcolormap operation.
 */

struct cg2pr {
     struct    cg2fb *cgpr_va;
     int       cgpr_fd;
     int       cgpr_planes;        /* Color bit plane mask register */
     struct    pr_pos cgpr_offset;
};

#define cg2_d(pr) ((struct cg2pr *)(pr)->pr_data)
#define cg2_fbfrompr(pr) (((struct cg2pr *)(pr)->pr_data)->cgpr_va)
#define cg2_ropword(cgd, plane, ax, ay)\
          (cg2_ropwordaddr((cgd)->cgpr_va, (plane),\
           (cgd)->cgpr_offset.x+(ax),(cgd)->cgpr_offset.y+(ay)) )
#define cg2_pixel(cgd, ax, ay)\
          (cg2_pixaddr((cgd)->cgpr_va,\
           (cgd)->cgpr_offset.x+(ax),(cgd)->cgpr_offset.y+(ay)) )
#define cg2_roppixel(cgd, ax, ay)\
          (cg2_roppixaddr((cgd)->cgpr_va,\
           (cgd)->cgpr_offset.x+(ax),(cgd)->cgpr_offset.y+(ay)) )
#define cg2_prd_skew(cgd, ax)\
           (((cgd)->cgpr_offset.x+(ax)) & 15)

extern    struct pixrectops cg2_ops;

int       cg2_rop();
int       cg2_putcolormap();
int       cg2_putattributes();

#ifndef KERNEL
int       cg2_stencil();
int       cg2_batchrop();
struct    pixrect *cg2_make();
int       cg2_destroy();
int       cg2_get();
int       cg2_put();
int       cg2_vector();
struct    pixrect *cg2_region();
int       cg2_getcolormap();
int       cg2_getattributes();
#endif !KERNEL
```

```
/*
 * (cgtwo.c) Sun2 (Memory Mapped) Color Board Driver
 * Copyright (c) 1984 by Sun Microsystems, Inc.
 */

/*
 * As a driver for a frame-buffer device, cgtwo.c must provide not only the
 * standard device-driver functionality, but also low-level support for the
 * Sun virtual desktop. That is to say, frame-buffer drivers not only
 * implement the standard device-driver hardware interface, but also declare,
 * initialize and export the pixrect structures which allow the kernel to
 * view the frame-buffer memory as a large rectangle within which it can
 * rapidly paint a cursor. As a consequence, some of the code here is pixrect
 * related, though among the muck you'll find the operations common to all
 * memory-mapped drivers.
 *
 * The kernel does not context switch frame buffers, despite the fact that some
 * of them (including the Sun2 Color Board which this driver controls) do have
 * context. In general, the kernel assumes that frame buffers either have no
 * context that needs to be switched, or are used in a manner that doesn't
 * require them to be context switched. SunWindows takes the second of these
 * tacks, arbitrating frame-buffer access (with pixwin locking) so that no
 * process can use the frame buffer while another process has "context" in it.
 *
 */

#include "cgtwo.h"          /* installed device count --from config */
#include "win.h"
#if NCGTWO > 0

#include <sys/param.h>          /* general kernel parameters */
#include <sys/buf.h>            /* I/O buffers */
#include <sys/errno.h>          /* system error reporting */
#include <sys/ioctl.h>          /* ioctl definitions */
#include <sys/map.h>            /* resource allocation maps */
#include <sys/vmmac.h>          /* virtual memory related conversion macros */

/* <machine> is a symbolic link to sun[234] */
#include <machine/pte.h>        /* page table entries */
#include <machine/mmu.h>        /* memory-management unit */
#include <machine/psl.h>        /* process status register */

#include <sun/fbio.h>           /* frame buffer definitions */

/* <sundev> is the device driver source directory */
#include <sundev/mbvar.h>       /* bus-interface definitions */

/* <pixrect> contains pixrect-related source */
#include <pixrect/pixrect.h>  /* basic pixrect definitions */
#include <pixrect/pr_impl_util.h>  /* pixrect utilities */
#include <pixrect/memreg.h>     /* rasterop hardware registers */
#include <pixrect/cg2reg.h>     /* Sun2 color frame buffer definitions */
#include <pixrect/cg2var.h>     /* more Sun2 color frame buffer */
```

```
/* probe size in bytes -- enough for the useful part of the board */
#define CG2_PROBESIZE CG2_MAPPED_SIZE

/* Mainbus device data */
int cgtwoprobe(), cgtwoattach();

struct mb_device *cgtwoinfo[NCGTWO];
struct mb_driver cgtwodriver = {
     cgtwoprobe, 0, cgtwoattach, 0, 0, 0,
     CG2_PROBESIZE, "cgtwo", cgtwoinfo, 0, 0, 0, 0
};

/* Driver per-unit data */
struct cg2_softc {
     int flags;      /* misc. flags; bits defined in cg2var.h */
                     /* (struct cg2pr, flags member) */
     struct cg2fb *fb;    /* virtual address */
     int w, h;       /* resolution */
#if NWIN > 0
     Pixrect pr;          /* kernel pixrect and private data */
     struct cg2pr prd;
#endif NWIN > 0
} cg2_softc[NCGTWO];

/* default structure for FBIOGATTR/FBIOGTYPE ioctls */
static struct fbgattr fbgattr_default =  {
/* real_type owner */
     FBTYPE_SUN2COLOR, 0,
/* fbtype: type h w depth cms size */
     { FBTYPE_SUN2COLOR, 0, 0, 8,    256,  CG2_MAPPED_SIZE },
/* fbsattr: flags emu_type */
     { FB_ATTR_DEVSPECIFIC, -1,
/* dev_specific: FLAGS, BUFFERS, PRFLAGS */
     { FB_ATTR_CG2_FLAGS_PRFLAGS, 1, 0 } },
/* emu_types */
     { -1, -1, -1, -1}
};

/* Double buffering enable flag */
int cg2_dblbuf_enable = 1;

#if NWIN > 0

/* SunWindows specific stuff */

/* kernel pixrect ops vector */
static struct pixrectops pr_ops = {
     cg2_rop,
     cg2_putcolormap,
     cg2_putattributes
};
#endif NWIN > 0
```

```
cgtwoprobe(reg, unit)
     caddr_t reg;
     int unit;
{
     register struct cg2fb *fb = (struct cg2fb *) reg;
     register struct cg2_softc *softc;

     /*
      * Check if board is present and strapped for 2M decoding.
      * If this fails, remap for 4M decoding and try again.
      */
     if (probeit(fb)) {
          fbmapin((caddr_t) fb, fbgetpage((caddr_t) fb) +
               (int) btop(CG2_MAPPED_OFFSET), CG2_MAPPED_SIZE);

          if (probeit(fb))
               return 0;
     }

     softc = &cg2_softc[unit];
     softc->fb = fb;
     softc->flags = 0;

     /* check for supported resolution */
     switch (fb->status.reg.resolution) {
     case CG2_SCR_1152X900:
          softc->w = 1152;
          softc->h =  900;
          softc->flags = CG2D_STDRES;
          break;
     case CG2_SCR_1024X1024:
          softc->w = 1024;
          softc->h = 1024;
          break;
     default:
          printf("%s%d: unsupported resolution (%d)0,
               cgtwodriver.mdr_cname, unit,
               fb->status.reg.resolution);
          return 0;
     }

     return CG2_PROBESIZE;
}

static
probeit(fb)
     register struct cg2fb *fb;
{
     union {
          struct cg2statusreg reg;
          short word;
     } status;
```

```
#define   allrop(fb, reg)      ((short *)  &(fb)->ropcontrol[CG2_ALLROP].ropregs.reg)
#define   pixel0(fb)      ((char *)  &fb->ropio.roppixel.pixel[0][0])

     /*
      * Probe sequence:
      *
      * set board for pixel mode access
      * enable all planes
      * set rasterop function to CG_SRC
      * disable end masks
      * set fifo shift/direction to zero/left-to-right
      * write 0xa5 to pixel at (0,0)
      * check pixel value
      * enable subset of planes (0xcc)
      * set rasterop function to ~CG_DEST
      * write to pixel at (0,0) again
      * enable all planes again
      * read pixel value; should be 0xa5 ^ 0xcc = 0x69
      */
     status.word = peek(&fb->status.word);
     status.reg.ropmode = SWWPIX;
     if (poke(&fb->status.word, status.word) ||
          poke((short *) &fb->ppmask.reg, 255) ||
          poke(allrop(fb, mrc_op), CG_SRC) ||
          poke(allrop(fb, mrc_mask1), 0) ||
          poke(allrop(fb, mrc_mask2), 0) ||
          poke(allrop(fb, mrc_shift), 1 << 8) ||
          pokec(pixel0(fb), 0xa5) ||
          pokec(pixel0(fb), 0) ||
          peekc(pixel0(fb)) != 0xa5 ||
          poke((short *) &fb->ppmask.reg, 0xcc) ||
          poke(allrop(fb, mrc_op), ~CG_DEST) ||
          pokec(pixel0(fb), 0) ||
          poke((short *) &fb->ppmask.reg, 255) ||
          peekc(pixel0(fb)) != (0xa5 ^ 0xcc))
          return 1;

     return 0;

#undef    allrop
#undef    pixel0
}


cgtwoattach(md)
     struct mb_device *md;
{
     register struct cg2_softc *softc = &cg2_softc[md->md_unit];
     register struct cg2fb *fb = softc->fb;
     register int flags = softc->flags;

#define   dummy     flags
```

```
/* set interrupt vector */
if (md->md_intr)
     fb->intrptvec.reg = md->md_intr->v_vec;
else
     printf("WARNING: no interrupt vector specified in config file0);

/*
 * Determine whether this is a Sun-2 or Sun-3 color board
 * by setting the wait bit in the double buffering register
 * and seeing if it clears itself during retrace.
 *
 * On the Sun-2 color board this just writes a bit in the
 * "wordpan" register.
 */
fb->misc.nozoom.dblbuf.word = 0;
fb->misc.nozoom.dblbuf.reg.wait = 1;

/* wait for leading edge, then trailing edge of retrace */
while (fb->status.reg.retrace)
     /* nothing */ ;
while (!fb->status.reg.retrace)
     /* nothing */ ;
while (fb->status.reg.retrace)
     /* nothing */ ;

if (fb->misc.nozoom.dblbuf.reg.wait) {
     /* Sun-2 color board */
     fb->misc.nozoom.dblbuf.reg.wait = 0;
     flags |= CG2D_ZOOM;
}
else {
     /* Sun-3 color board (or better) */
     flags |= CG2D_32BIT | CG2D_NOZOOM;

     if (fb->status.reg.fastread)
          flags |= CG2D_FASTREAD;

     if (fb->status.reg.id)
          flags |= CG2D_ID | CG2D_ROPMODE;

     /*
      * Probe for double buffering feature.
      * Write distinctive values to one pixel in both buffers,
      * then two pixels in buffer B only.
      * Read from buffer B and see what we get.
      *
      * Warning: assumes we were called right after cgtwoprobe
      */
     cg2_setfunction(fb, CG2_ALLROP, CG_SRC);
     fb->ropio.roppixel.pixel[0][0] = 0x5a;
     fb->ropio.roppixel.pixel[0][0] = 0xa5;
     fb->misc.nozoom.dblbuf.reg.nowrite_a = 1;
     fb->ropio.roppixel.pixel[0][0] = 0xc3;
```

```
        fb->ropio.roppixel.pixel[0][4] = dummy;
        if (fb->ropio.roppixel.pixel[0][0] == 0x5a) {
            fb->misc.nozoom.dblbuf.reg.read_b = 1;

            if (fb->ropio.roppixel.pixel[0][0] == 0xa5 &&
                fb->ropio.roppixel.pixel[0][4] == 0xc3 &&
                cg2_dblbuf_enable)
                flags |= CG2D_DBLBUF;
        }
        fb->misc.nozoom.dblbuf.word = 0;
    }

    softc->flags = flags;

#ifndef sun2
    /* re-map into correct VME space if necessary */
    {
        int page = fbgetpage((caddr_t) fb);

        if (((flags & CG2D_32BIT) != 0) !=
            ((page & PGT_MASK) == PGT_VME_D32))
            fbmapin((caddr_t) fb,
                page ^ (PGT_VME_D16 ^ PGT_VME_D32),
                CG2_MAPPED_SIZE);
    }
#endif !sun2

    /* print informative message */
    printf("%s%d: Sun-%c color board%s%s0,
        md->md_driver->mdr_dname, md->md_unit,
        flags & CG2D_ZOOM ? '2' : '3',
        flags & CG2D_DBLBUF ? ", double buffered" : "",
        flags & CG2D_FASTREAD ? ", fast read" : "");
}

cgtwoopen(dev, flag)
    dev_t dev;
    int flag;
{
    return fbopen(dev, flag, NCGTWO, cgtwoinfo);
}

/*ARGSUSED*/
cgtwoclose(dev, flag)
    dev_t dev;
{
    register struct cg2_softc *softc = &cg2_softc[minor(dev)];
    register struct cg2fb *fb = softc->fb;

    /* fix up zoom and/or double buffering on close */

    if (softc->flags & CG2D_ZOOM) {
        fb->misc.zoom.wordpan.reg = 0;      /* hi pixel adr = 0 */
```

```
		fb->misc.zoom.zoom.word = 0;  /* zoom=0, yoff=0 */
		fb->misc.zoom.pixpan.word = 0;      /* pix adr=0, xoff=0 */
		fb->misc.zoom.varzoom.reg = 255; /* unzoom at line 4*255 */
	}

	if (softc->flags & CG2D_NOZOOM)
		fb->misc.nozoom.dblbuf.word = 0;

	return 0;
}

cgtwommap(dev, off, prot)
	dev_t dev;
	off_t off;
	int prot;
{
	return fbmmap(dev, off - CG2_MAPPED_OFFSET,
		prot, NCGTWO, cgtwoinfo, CG2_MAPPED_SIZE);
}

/*ARGSUSED*/
cgtwoioctl(dev, cmd, data, flag)
	dev_t dev;
	int cmd;
	caddr_t data;
	int flag;
{
	register struct cg2_softc *softc = &cg2_softc[minor(dev)];

	switch (cmd) {

	case FBIOGTYPE: {
		register struct fbtype *fbtype = (struct fbtype *) data;

		*fbtype = fbgattr_default.fbtype;
		fbtype->fb_height = softc->h;
		fbtype->fb_width  = softc->w;
	}
	break;

	case FBIOGATTR: {
		register struct fbgattr *gattr = (struct fbgattr *) data;

		*gattr = fbgattr_default;
		gattr->fbtype.fb_height = softc->h;
		gattr->fbtype.fb_width = softc->w;

		if (softc->flags & CG2D_NOZOOM)
			gattr->sattr.dev_specific[FB_ATTR_CG2_FLAGS] |=
				FB_ATTR_CG2_FLAGS_SUN3;

		if (softc->flags & CG2D_DBLBUF)
			gattr->sattr.dev_specific[FB_ATTR_CG2_BUFFERS] = 2;
```

```
          gattr->sattr.dev_specific[FB_ATTR_CG2_PRFLAGS] = softc->flags;
     }
     break;

     case FBIOSATTR:
          break;

#if NWIN > 0

     case FBIOGPIXRECT:
          ((struct fbpixrect *) data)->fbpr_pixrect = &softc->pr;

          /* initialize pixrect */
          softc->pr.pr_ops = &pr_ops;
          softc->pr.pr_size.x = softc->w;
          softc->pr.pr_size.y = softc->h;
          softc->pr.pr_depth = CG2_DEPTH;
          softc->pr.pr_data = (caddr_t) &softc->prd;

          /* initialize private data */
          bzero((char *) &softc->prd, sizeof softc->prd);
          softc->prd.cgpr_va = softc->fb;
          softc->prd.cgpr_fd = 0;
          softc->prd.cgpr_planes = 255;
          softc->prd.ioctl_fd = minor(dev);
          softc->prd.flags = softc->flags;
          softc->prd.linebytes = softc->w;

          /* enable video */
          softc->fb->status.reg.video_enab = 1;

          break;

#endif NWIN > 0

     /* get info for GP */
     case FBIOGINFO: {
          register struct fbinfo *fbinfo = (struct fbinfo *) data;

          fbinfo->fb_physaddr =
               (fbgetpage((caddr_t) softc->fb) << PGSHIFT) -
               CG2_MAPPED_OFFSET & 0xffffff;
          fbinfo->fb_hwwidth = softc->w;
          fbinfo->fb_hwheight = softc->h;
          fbinfo->fb_ropaddr = (u_char *) softc->fb;
     }
     break;

     /* set video flags */
     case FBIOSVIDEO:
          softc->fb->status.reg.video_enab =
               (* (int *) data) & FBVIDEO_ON ? 1 : 0;
          break;
```

```
	/* get video flags */
	case FBIOGVIDEO:
		* (int *) data = softc->fb->status.reg.video_enab
			? FBVIDEO_ON : FBVIDEO_OFF;
		break;

	case FBIOVERTICAL:
		cgtwo_wait(minor(dev));
		break;

	default:
		return ENOTTY;
	}

	return 0;
}

/* wait for vertical retrace interrupt */
cgtwo_wait(unit)
	int unit;
{
	register struct mb_device *md = cgtwoinfo[unit & 255];
	register struct cg2_softc *softc = &cg2_softc[unit & 255];
	int s;

	if (md->md_intr == 0)
		return;

	s = splx(pritospl(md->md_intpri));
	softc->fb->status.reg.inten = 1;
	(void) sleep((caddr_t) softc, PZERO - 1);
	(void) splx(s);
}

/* vertical retrace interrupt service routine */
cgtwointr(unit)
	int unit;
{
	register struct cg2_softc *softc = &cg2_softc[unit];

	softc->fb->status.reg.inten = 0;
	wakeup((caddr_t) softc);

#ifdef lint
	cgtwointr(unit);
#endif
}
```

```
/*
 * (fbutil.c) Frame Buffer Driver Support Utilities
 * Copyright (c) 1985, 1987 by Sun Microsystems, Inc.
 */

/*
 * The routines in this file, called from many the Sun frame buffer drivers,
 * perform the essential operations necessary for all memory-mapped drivers.
 */

#include <sys/param.h>        /* machine dependent kernel parameters */
#include <sys/buf.h>          /* I/O buffers */
#include <sys/errno.h         /* System error reporting */
#include <sys/mman.h>         /* Memory-mapping definitions */
#include <sys/vmmac.h>        /* Virtual memory related conversion macros */

/* <machine> is a symbolic link set to sun[234] */
#include <machine/pte.h>      /* page table entries */

/* <sundev> is the device driver source directory */
#include <sundev/mbvar.h>     /* bus-interface definitions */

/*
 * Makes the necessary error checks and then returns. Everything is OK if the
 * device is predefined in the config file and if the probe routine found it as
 * expected.
 */
int fbopen(dev, flag, numdevs, mb_devs)
     dev_t dev;
     int flag, numdevs;
     struct mb_device **mb_devs;
{
     register struct mb_device *md;

     if (minor(dev) >= numdevs ||
          (md = mb_devs[minor(dev)]) == 0 ||
          md->md_alive == 0)
          return ENXIO;

     return 0;
}

/*
 * Work from the device address and an offset within its address
 * space to get the page frame number for the page to be mapped.
 */
int fbmmap(dev, off, prot, numdevs, mb_devs, size)
     dev_t dev;
     off_t off;
     int rot;
     int numdevs;
     struct mb_device **mb_devs;
     int size;
```

```
{
	if ((u_int) off >= size)
		return -1;

	return fbgetpage(mb_devs[minor(dev)]->md_addr + off);

}

/* Get page frame number and page type */
fbgetpage(addr)
	caddr_t addr;
{
	return (int) hat_getkpfnum((addr_t) addr);
}

/*
 * Simplified mapin and mapout. Note that, since these
 * routines are implemented in terms of Usrptmap (which has been
 * preserved for compatibility reasons) they will work with either SunOS
 * release 4.0 or with earlier releases.
fbmapin(virt, phys, size)
	caddr_t virt;
	int phys;
	int size;
{
	mapin(&Usrptmap[btokmx((struct pte *) virt)], btop(virt),
		(u_int) phys, btoc(size), PG_V | PG_KW);
}

fbmapout(virt, size)
	caddr_t virt;
	int size;
{
	mapout(&Usrptmap[btokmx((struct pte *) virt)], btoc(size));
}

#ifdef sun2
/*
 * Some Sun-2 frame-buffer devices allowed the user to enable/disable interrupts, and
 * even to change the interrupt level. Thus, fbintr is necessary so that the
 * kernel will always be able to find the interrupting device. If fbintr finds
 * an interrupting device, it returns with a 1 after calling intclear to turn
 * off its interrupt.
 */
fbintr(numdevs, mb_devs, intclear)
	int numdevs;
	register struct mb_device **mb_devs;
	int (*intclear)();
{
	register struct mb_device *md;

	while (--numdevs >= 0)
```

```
          if ((md = *mb_devs++) &&
               md->md_alive &&
               (*intclear)(md->md_addr))
               return 1;

     return 0;
}
#endif sun2
```

E.3. Sky Floating-Point Driver

```
/*
 * (skyreg.h) Sky Floating Point Processor Registers
 * Copyright (c) 1983 by Sun Microsystems, Inc.
 */

struct    skyreg {
     u_short   sky_command;
     u_short   sky_status;
     union {
          short     skyu_dword[2];
          long skyu_dlong;
     } skyu;
#define   sky_data  skyu.skyu_dlong
#define   sky_dlreg skyu.skyu_dword[0]
     long sky_ucode;
     u_short   sky_vector;      /* VME interrupt vector number */
};

/* command masks */
#define   SKY_SAVE       0x1040
#define   SKY_RESTOR     0x1041
#define   SKY_NOP        0x1063
#define   SKY_START0     0x1000
#define   SKY_START1     0x1001

/* status masks */
#define   SKY_IHALT      0x0000
#define   SKY_INTRPT     0x0003
#define   SKY_INTENB     0x0010
#define   SKY_RUNENB     0x0040
#define   SKY_SNGRUN     0x0060
#define   SKY_RESET      0x0080
#define   SKY_IODIR      0x2000
#define   SKY_IDLE       0x4000
#define   SKY_IORDY      0x8000
```

```
/*
 * (sky.c) SKY Floating-point Processor Driver
 * Copyright (c) 1985 by Sun Microsystems, Inc.
 */

/*
 * The Sky driver is quite unusual in that maintains some state information
 * in the kernel user structure. This is because the kernel must context
 * switch the Sky board among the processes that wish to use it. This is not
 * typical, and, in fact, there is currently no way for users to add new
 * devices which, like the Sky board, must be context switched by the kernel.
 *
 * The Sky board is used only with Sun2 machines, and machines with Sky boards
 * are known to have only one installed.
 */

/*
 * Most device drivers include about the same set of system header files,
 * with variation reflecting driver differences in functionality. The system
 * include files are located in directories whose location is fixed relative
 * to the configuration directories (for both source and object distributions.)
 * Note that there is not a sky.h file included here; the sky board is a
 * special case and we know that there's only one installed.
 */

#include <sys/param.h>          /* general kernel parameters */
#include <sys/buf.h>            /* I/O buffers */
#include <sys/file.h>           /* open file information */
#include <sys/dir.h>            /* file system directories */
#include <sys/user.h>           /* kernel per-process status */

/* <machine> is a symbolic link set to either sun2 or sun3 */
#include <machine/pte.h>        /* page table entries */
#include <machine/mmu.h>        /* memory management unit */
#include <machine/cpu.h>        /* architecture-related defs */
#include <machine/scb.h>        /* system control block */

/* ../sundev is the device driver source directory */
#include <sundev/mbvar.h>       /* bus interface definitions */
#include <sundev/skyreg.h>      /* sky register definitions */

/*
 * The "page" size (for the VME sky board only) is an offset which must be
 * added to the device base address to get access to the full set of device
 * registers. The second page (page 1) is taken as the supervisor page and
 * allows access to all the registers; the first (0) page is the user page and
 * does not, thus preventing access to the registers needed to load microcode
 * and context switch the device. In user mode it's only possible to access the
 * registers needed to control floating-point operations.
 */
#define   SKYPGSIZE 0x800

/* auto-configuration information */
```

```
int       skyprobe(), skyattach(), skyintr();
struct    mb_device *skyinfo[1];    /* Only one Sky board */
struct    mb_driver skydriver = {
     skyprobe, 0, skyattach, 0, 0, skyintr,
     2 * SKYPGSIZE, "sky", skyinfo, 0, 0, 0, 0,
};

/*
 * The global variable skyaddr is set in skyprobe to contain the
 * base address of the "supervisor page" (page 1) of the Sky board (the base
 * address of the device registers.)
 */
struct    skyreg *skyaddr;

/*
 * These two global variables are used to control extraordinary aspects of the
 * Sky driver logic:
 *      skyinit is set to 1 when the device (during system initialization)
 * is opened for microcode loading. When the microcode loader closes the
 * device, skyinit is set to 2, indicating that the device is available
 * for general use. This mechanism is necessary to handle the special open
 * state needed for microcode loading.
 *      skyisnew is even more peculiar, being necessary only to distinguish
 * two slightly different versions of the Sky board.
 */
int skyinit = 0, skyisnew = 0;

/*ARGSUSED*/
skyprobe(reg, unit)
     caddr_t reg;
     int unit;
{
     register struct skyreg *skybase = (struct skyreg *)reg;

     /* Is something there? */
     if (peek((short *)skybase) == -1)
          return (0);

     /* If so, is it a Sky board? */
     if (poke((short *)&skybase->sky_status, SKY_IHALT))
          return (0);

     skyaddr = (struct skyreg *)(SKYPGSIZE + reg);
     if (cpu == CPU_SUN2_120 ||
         poke((short *)&skyaddr->sky_status, SKY_IHALT)) {

          /* old VMEbus or Multibus version of the Sky board */
          skyaddr = (struct skyreg *)reg;
          skyisnew = 0;
     } else
          skyisnew = 1;

     return (sizeof (struct skyreg));
```

```
}

/*
 * If it's the new version of the board, then it has to be told what interrupt
 * to respond to. This is true for both vectored and auto-vectored interrupts.
 * In the auto-vectored case, the VME interrupt vector is set to be identical
 * to the 68000 auto-vector for the appropriate interrupt level. For the old
 * version of the Sky board, skyattach does nothing.
 */
skyattach(md)
     struct mb_device *md;
{

     if (skyisnew) {
          if (!md->md_intr) {
               /* auto-vectored interrupts */
               (void) poke((short *)&skyaddr->sky_vector,
                   AUTOBASE + md->md_intpri);
          } else {
               /* vectored interrupts */
               (void) poke((short *)&skyaddr->sky_vector,
                   md->md_intr->v_vec);
          }
     }
}

/*ARGSUSED*/
skyopen(dev, flag)
     dev_t dev;
     int flag;
{
     int i;
     register struct skyreg *s = skyaddr;

     if (skyaddr == 0) /* skyprobe didn't find the device */
          return (ENXIO);

     if (skyinit == 2) {
          /*
           * skyinit is 2 only when skyclose has previously been
           * called. This is true only in the case where skyclose was
           * called by the microcode loader, and so it's used here to recognize
           * the first time that the device is opened for use by a user
           * process. Thus, it's here that the device (and its related
           * bookkeeping fields) need to be initialized.
           */
          s->sky_status = SKY_RESET;
          s->sky_command = SKY_START0;
          s->sky_command = SKY_START0;
          s->sky_command = SKY_START1;
          s->sky_status = SKY_RUNENB;
          u.u_skyctx.usc_used = 1;
          u.u_skyctx.usc_cmd = SKY_NOP;
```

```
		for (i=0; i<8; i++)
			u.u_skyctx.usc_regs[i] = 0;
		skyrestore();

	} else if (flag & FNDELAY)
		/*
		 * This open is for the the user program that loads the microcode.
		 * This is a special case that allows it to open the device, even
		 * though it isn't initialized.
		 */
		skyinit = 1;

	else
		return (ENXIO);
	return (0);
}

/*ARGSUSED*/
skyclose(dev, flag)
	dev_t dev;
	int flag;
{

	/*
	 * Call skysave in case a user aborted and left the board in an
	 * unclean state. We're really not saving the device state here, but
	 * rather calling skysave to ensure that the state is safe for the
	 * next user.
	 */
	if (skyinit == 2)
		skysave();

	/*
	 * This is not the normal case. skyinit is being set to 2 to indicate to
	 * skyopen that the device has been initialized.
	 */
	if (skyinit == 1)
		skyinit = 2;
	u.u_skyctx.usc_used = 0;
	return (0);
}

/*ARGSUSED*/
skymmap(dev, off, prot)
	dev_t dev;
	off_t off;
	int prot;
{

	if (off)
		return (-1);

	/*
```

```
     * If this is a VME Sky board, and the board has been initialized (its
     * microcode loaded), then allow the user process to have access only to
     * the "user" page. This allows users to do floating-point operations,
     * but not to load microcode. The Multibus Sky board doesn't offer such
     * protection, so any process can load microcode and screw up other users
     * of the board. If this is a VME board, but we've still in the
     * microcode-loading state, allow access to the "supervisor" version of
     * the registers so we can load the microcode.
     */
     off = (off_t)skyaddr;
     if (skyisnew && skyinit == 2)          /* use user page */
          off -= SKYPGSIZE;

     return (hat_getkpfnum((addr_t) off));
}

/*
 * skyintr is also quite atypical, being used only for error reporting
 * and to disable interrupts. It must disable interrupts because they may (on
 * the Multibus version for sure) have been accidently set by a user process
 * with access to the device registers. The kernel must be able to handle
 * all the interrupts which can be generated by all the devices, even if it
 * doesn't use them for anything.
 */
/*ARGSUSED*/
skyintr(n)
     int n;
{
        static u_short  skybooboo = 0;

        if (skyaddr && (skyaddr->sky_status & (SKY_INTENB|SKY_INTRPT))) {
               if (skyaddr->sky_status & SKY_INTENB) {
                    printf("skyintr: sky board interrupt enabled, status = 0x%x\n",
                                skyaddr->sky_status);
                        skyaddr->sky_status &= ~(SKY_INTENB|SKY_INTRPT);
                        return (1);
               }
               if (!skybooboo && (skyaddr->sky_status & SKY_INTRPT)) {
                    printf("skyintr: sky board unrecognized status, status = 0x%x\n",
                                skybooboo = skyaddr->sky_status);
                        return (0);
               }
     }
     return (0);
}

/*
 * skysave does the actual work of saving the device state. It has to
 * jump through some hoops to do so, but these hoops are completely device
 * specific.
 */
skysave()
{
        register short i;
        register struct skyreg *s = skyaddr;
        register u_short stat;

        for (i = 0; i < 100; i++) {
                stat = s->sky_status;
                if (stat & SKY_IDLE) {
                        u.u_skyctx.usc_cmd = SKY_NOP;
                        goto sky_save;
                }
                if (stat & SKY_IORDY)
                        goto sky_ioready;
        }
        printf("sky0: hung\n");
        skyinit = 0;
        u.u_skyctx.usc_used = 0;
        return;

         /* I/O is ready, is it a read or write? */
sky_ioready:
        s->sky_status = SKY_SNGRUN;     /* set single step mode */
        if (stat & SKY_IODIR)
                i = s->sky_d1reg;
        else
                s->sky_d1reg = i;

        /*
         * Check again since data may have been in a long word.
         */
        stat = s->sky_status;
        if (stat & SKY_IORDY)
                if (stat & SKY_IODIR)
                        i = s->sky_d1reg;
                else
                        s->sky_d1reg = i;

        /*
         * Read and save the command register. Decrement it by 1 since it's
         * actually Sky program counter and must be backed up.
         */
        u.u_skyctx.usc_cmd = s->sky_command - 1;

        /*
         * Reinitialize the board.
         */
        s->sky_status = SKY_RESET;
        s->sky_command = SKY_START0;
        s->sky_command = SKY_START0;
        s->sky_command = SKY_START1;
        s->sky_status = SKY_RUNENB;

        /*
         * Do the actual context save. (Unrolled loop for efficiency.)
```

```
	 */
sky_save:
	s->sky_command = SKY_NOP;	/* set device to a clean mode */
	s->sky_command = SKY_SAVE;
	u.u_skyctx.usc_regs[0] = s->sky_data;
	u.u_skyctx.usc_regs[1] = s->sky_data;
	u.u_skyctx.usc_regs[2] = s->sky_data;
	u.u_skyctx.usc_regs[3] = s->sky_data;
	u.u_skyctx.usc_regs[4] = s->sky_data;
	u.u_skyctx.usc_regs[5] = s->sky_data;
	u.u_skyctx.usc_regs[6] = s->sky_data;
	u.u_skyctx.usc_regs[7] = s->sky_data;
}

skyrestore()
{
	register struct skyreg *s = skyaddr;

	if (skyinit != 2) {
		u.u_skyctx.usc_used = 0;
		return;
	}
	s->sky_command = SKY_NOP;	/* set device to a clean mode */

	/*
	 * Do the actual context restore.
	 */
	s->sky_command = SKY_RESTOR;
	s->sky_data = u.u_skyctx.usc_regs[0];
	s->sky_data = u.u_skyctx.usc_regs[1];
	s->sky_data = u.u_skyctx.usc_regs[2];
	s->sky_data = u.u_skyctx.usc_regs[3];
	s->sky_data = u.u_skyctx.usc_regs[4];
	s->sky_data = u.u_skyctx.usc_regs[5];
	s->sky_data = u.u_skyctx.usc_regs[6];
	s->sky_data = u.u_skyctx.usc_regs[7];
	s->sky_command = u.u_skyctx.usc_cmd;
}
```

E.4. Versatec Interface Driver

```
/*
 * (vcmd.h) Include file for user programs that'll give ioctl commands to the
 * Ikon 10071-5 Multibus/Versatec interface.
 * Copyright (c) 1983 by Sun Microsystems, Inc.
 */

#ifndef _IOCTL_
#include <sys/ioctl.h>
#endif

#define   VPRINT        0100
#define   VPLOT         0200
#define   VPRINTPLOT    0400
#define   VPC_TERMCOM   0040
#define   VPC_FFCOM     0020
#define   VPC_EOTCOM    0010
#define   VPC_CLRCOM    0004
#define   VPC_RESET     0002

/*
 * _IOR and _IOW encode read/write instructions to the kernel within the ioctl
 * command code. These instructions cause the kernel to read the ioctl
 * command argument into user space (_IOR), or to write it into kernel space (_IOW).
 */
#define   VGETSTATE _IOR(v, 0, int)
#define   VSETSTATE _IOW(v, 1, int)
```

```
/*
 * (vpreg.h) Registers for Ikon 10071-5 Multibus/Versatec interface.
 * Copyright (c) 1983 by Sun Microsystems, Inc.
 */

/*
 * Note that the vpdevice structure actually spans the registers of several
 * contiguous IC devices (a 8259 and a 8237.)  Only the low byte of each
 * (short) word is used.
 */

struct vpdevice {
     u_short   vp_status;       /* 00: mode(w) and status(r) */
     u_short   vp_cmd;          /* 02: special command bits(w) */
     u_short   vp_pioout;       /* 04: PIO output data(w) (unused) */
     u_short   vp_hiaddr;       /* 06: hi word of Multibus DMA address(w) */
     u_short   vp_icad0;        /* 08: ad0 of 8259 interrupt controller */
     u_short   vp_icad1;        /* 0A: ad1 of 8259 interrupt controller */

     /* The rest of the fields are for the 8237 DMA controller */
     u_short   vp_addr;         /* 0C: DMA word address */
     u_short   vp_wc;           /* 0E: DMA word count */
     u_short   vp_dmacsr;       /* 10: command and status (unused) */
     u_short   vp_dmareq;       /* 12: request (unused) */
     u_short   vp_smb;          /* 14: single mask bit (unused) */
     u_short   vp_mode;         /* 16: dma mode */
     u_short   vp_clrff;        /* 18: clear first/last flip-flop */
     u_short   vp_clear;        /* 1A: DMA master clear */
     u_short   vp_clrmask;      /* 1C: clear mask register */
     u_short   vp_allmask;      /* 1E: all mask bits (unused) */
};

/*
 * Warning - this is one of those devices in which the read bits are not
 * identical to write bits.
 */

/* vp_status bits (read) */
#define   VP_IS8237  0x80       /* 1 if 8237 (sanity checker) */
#define   VP_REDY    0x40       /* printer ready */
#define   VP_DRDY    0x20       /* printer and interface ready */
#define   VP_IRDY    0x10       /* interface ready */
#define   VP_PRINT   0x08       /* print mode */
#define   VP_NOSPP   0x04       /* not in SPP mode */
#define   VP_ONLINE  0x02       /* printer online */
#define   VP_NOPAPER 0x01       /* printer out of paper */

/* vp_status bits (write) */
#define   VP_PLOT    0x02       /* enter plot mode */
#define   VP_SPP     0x01       /* enter SPP mode */

/* vp_cmd bits */
#define   VP_RESET   0x10       /* reset the plotter and interface */
```

```
#define    VP_CLEAR    0x08        /* clear the plotter */
#define    VP_FF       0x04        /* form feed to plotter */
#define    VP_EOT      0x02        /* EOT to plotter */
#define    VP_TERM     0x01        /* line terminate to plotter */

/* vp_mode bits */
#define    VP_DMAMODE 0x47         /* put interface in DMA mode */

/*
 * These two values are used to set the device (which is reticent to disclose
 * that it has issued an interrupt) so that the polling routine can find out.
 */
#define    VP_ICPOLL 0x0C
#define    VP_ICEOI  0x20
```

```
/*
 * (vp.c) DMA driver for Ikon 10071-5 Versatec matrix printer/plotter driver.
 * Copyright (c) 1985 by Sun Microsystems, Inc.
 */

/*
 * Most device drivers include about the same set of system header files, with
 * variation reflecting driver differences in functionality. The system include
 * files are located in directories whose location is fixed relative to the
 * configuration directories (for both source and object distributions.) vp.h
 * is presumed to be in the configuration directory, where config will have
 * left it and from which it is assumed that driver source files (like this one)
 * are compiled.
 */

#include "vp.h"                 /* installed device count -- from config */
#include <sys/param.h>          /* general kernel parameters */
#include <sys/dir.h>            /* file system directories */
#include <sys/user.h>           /* kernel per-process status */
#include <sys/buf.h>            /* I/O buffers */
#include <sys/systm.h>          /* miscellaneous kernel variables */
#include <sys/kernel.h>         /* kernel global variables */
#include <sys/map.h>            /* resource allocation maps */
#include <sys/ioctl.h>          /* ioctl definitions */
#include <sys/vcmd.h>           /* for all Versatec interface drivers */
#include <sys/uio.h>            /* uio structures */

/* <machine> is a symbolic link set to either sun2 or sun3 */
#include <machine/psl.h>        /* processor status codes */
#include <machine/mmu.h>        /* memory-management unit */

/* <sundev> is the device driver source directory */
#include <sundev/vpreg.h>       /* vp register definitions */
#include <sundev/mbvar.h>       /* bus-interface definitions */

/*
 * Define the Versatec sleeping priority to be lower than PZERO, that is, make
 * its sleep be uninterruptible by signals. This is appropriate because the
 * events which we'll be waiting for, slow as they may be, are relatively fast
 * and sure (unlike user input) to occur.
 */
#define   VPPRI     (PZERO-1)

/*
 * Define an array of vp_softc structures, one for each of the NVP
 * installed devices. By convention, the names xx_softc and
 * xx_device are used for the private, per-device software state
 * structure.
 */
struct vp_softc {
     int   sc_state;          /* current device state */
     struct buf *sc_bp;       /* buffer mapped to device */
     int   sc_mbinfo;         /* stash for mbsetup's return code */
```

```
} vp_softc[NVP];

/*
 * sc_state bits - passed in VGETSTATE and VSETSTATE ioctl calls.
 * The user-level ioctl command codes are in vcmd.h, normally found
 * in /usr/include/sys
 */
#define   VPSC_BUSY      0400000
#define   VPSC_MODE      0000700
#define   VPSC_SPP       0000400
#define   VPSC_PLOT      0000200
#define   VPSC_PRINT     0000100
#define   VPSC_CMNDS     0000076
#define   VPSC_OPEN      0000001

/* no special encoding in minor device number */
#define   VPUNIT(dev)    (minor(dev))

/*
 * Declare an array of private buf headers, by convention named rvpbuf, one for
 * each of the NVP installed devices.
 */
struct    buf  rvpbuf[NVP];

/* The autoconfig-related declarations. */
int vpprobe(), vpintr();
struct    mb_device *vpdinfo[NVP];
struct    mb_driver vpdriver = {
     vpprobe, 0, 0, 0, 0, vpintr,
     sizeof (struct vpdevice), "vp", vpdinfo, 0, 0, 0,
};

/*
 * vpprobe already indicates the persnickety nature of the device, a
 * nature that will become more clear as we proceed.
 */
vpprobe(reg)
     caddr_t reg;
{
     register struct vpdevice *vpaddr = (struct vpdevice *)reg;
     register int x;

     x = peek((short *)&vpaddr->vp_status);

     /*
      * Note that the device provides a sanity check bit, which
      * we can use to ensure that vpprobe is accurate
      */
     if (x == -1 || (x & VP_IS8237) == 0)
          return (0);

     /* Now reset the 8259; also return 0 if reset fails */
     if (poke((short *)&vpaddr->vp_cmd, VP_RESET))
```

```
                return (0);

        /*
         * Device-specific magic to shut up the device, by setting the 8259 -- it
         * doesn't have enough sense to wait for the driver's instructions, and
         * starts interrupting after being reset. Note that even this isn't
         * straightforward because of register write latency.
         */
        vpaddr->vp_icad0 = 0x12; /* ICW1, edge-trigger */
        DELAY(1);
        vpaddr->vp_icad1 = 0xFF; /* ICW2 - don't care (non-zero) */
        DELAY(1);
        vpaddr->vp_icad1 = 0xFE; /* IR0 - interrupt on DRDY edge */

        /* Also reset the 8237 */
        vpaddr->vp_clear = 1;

        return (sizeof (struct vpdevice));
}

vpopen(dev)
        dev_t dev;
{
        register struct vp_softc *sc;
        register struct mb_device *md;
        register int s;
        static int vpwatch = 0;

        /* Do a variety of error checks upon opening the device. Fail if dev
         * is greater than the configured number of devices, or if the device
         * (which is exclusive open) has already been opened, or if vpprobe
         * failed to find the device as expected.
         *
         * Note that, if the device wasn't found by the probe routine, both
         * vpdinfo[VPUNIT(dev)] and md->md_alive will be 0. Any given
         * driver may chose, for its convenience, to make either test, but it's
         * paranoid to -- as is done here -- make both. (All drivers have
         * access to md->md_alive; this isn't the case with xxdinfo).
         */
        if (VPUNIT(dev) >= NVP ||
            ((sc = &vp_softc[minor(dev)])->sc_state&VPSC_OPEN) ||
            (md = vpdinfo[VPUNIT(dev)]) == 0 || md->md_alive == 0)
                return (ENXIO);

        /*
         * vpwatch is a static local which is set to 0 the first time
         * vpopen is called. This code sets vpwatch to one and then
         * calls vptimo -- the effect is that vptimo gets called only once,
         * the first time a user process calls vpopen. But if you examine
         * vptimo, you'll see that it arranges matters so that it's called
         * repeatedly. This helps to keep the device from locking up.
         */
        if (!vpwatch) {
          vpwatch = 1;
          vptimo();
     }

     /*
      * Initialize softc state variable. Here we are, among other things, setting
      * sc->sc_state = VPSC_OPEN, which indicates that the device (which is
      * exclusive use) is tied up, and that no one else can open it. We are also
      * dispatching two commands, CLRCOM and VPC_RESET.
      */
     sc->sc_state = VPSC_OPEN|VPSC_PRINT | VPC_CLRCOM|VPC_RESET;

     /* Loop while any command is in process */
     while (sc->sc_state & VPSC_CMNDS) {
          /*
           * This critical section ensures that only one instance of the driver can
           * vpwait/vpcmd at any time. vpcmd clears command request
           * bits as it processes commands. This is absolutely necessary, since
           * vpcmd intends to actually dispatch a command (posted in
           * sc->sc_state) to the hardware.
           */
          s = splx(pritospl(md->md_intpri));
          vpwait(dev);
          vpcmd(dev);
          (void) splx(s);
     }
     return (0);
}

vpclose(dev)
     dev_t dev;
{
     register struct vp_softc *sc = &vp_softc[VPUNIT(dev)];

     sc->sc_state = 0;
}

vpstrategy(bp)
     register struct buf *bp;
{
     register struct vp_softc *sc = &vp_softc[VPUNIT(bp->b_dev)];
     register struct mb_device *md = vpdinfo[VPUNIT(bp->b_dev)];
     register struct vpdevice *vpaddr = (struct vpdevice *)md->md_addr;
     int s;
     int pa, wc;

     /*
      * The hardware doesn't support writes to odd addresses or DMA requests
      * of less than two bytes in length.
      */
     if (((int)bp->b_un.b_addr & 1) || bp->b_bcount < 2) {
          bp->b_flags |= B_ERROR;
          iodone(bp);
          return;
     }

     s = splx(pritospl(md->md_intpri));
     while (sc->sc_bp != NULL)
          sleep((caddr_t)sc, VPPRI);

     sc->sc_bp = bp;

     vpwait(bp->b_dev);
     /*Map next request for the now idle device onto the bus for a DMA transfer*/
     sc->sc_mbinfo = mbsetup(md->md_hd, bp, 0);

     vpaddr->vp_clear = 1;

     /* Get the address in DVMA space */
     pa = MBI_ADDR(sc->sc_mbinfo);

     /*
      * Now comes some VERY device-specific code, as we set the DMA transfer
      * address on the device.
      */
     vpaddr->vp_hiaddr = (pa >> 16) & 0xF;
     pa = (pa >> 1) & 0x7FFF;
     wc = (bp->b_bcount >> 1) - 1;
     bp->b_resid = 0;

     /*
      * Note the 2 sequential 8-bit writes into the same address to indicate
      * a 16-bit address!
      */
     vpaddr->vp_addr = pa & 0xFF;
     vpaddr->vp_addr = pa >> 8;

     vpaddr->vp_wc = wc & 0xFF;
     vpaddr->vp_wc = wc >> 8;
     vpaddr->vp_mode = VP_DMAMODE;
     vpaddr->vp_clrmask = 1;

     /*
      * By setting the VPSC_BUSY bit in sc->sc_state, we indicate that the device
      * is to sleep, and that vpwait is to loop. This is because we want to insure
      * that another command doesn't get issued until this DMA transfer is completed.
      */
     sc->sc_state |= VPSC_BUSY;

     (void) splx(s);          /* end of critical section */
}

/*
 * There is no read routine, as this is a write-only device.
 */
/*ARGSUSED*/
```

```
vpwrite(dev, uio)
     dev_t dev;
     struct uio *uio;
{

     if (VPUNIT(dev) >= NVP)
          return (ENXIO);
     return (physio(vpstrategy, &rvpbuf[VPUNIT(dev)], dev, B_WRITE,
              minphys, uio));
}

/*
 * vpwait kills time, but not by busy waiting. Instead, it relies on the
 * fact that sleep and wakeup aren't proper semaphores, and that ALL
 * processes which are sleeping on a channel wake when a wakeup is issued
 * on that channel. vpwait's sleep, then, is awaken by vpintr.
 */
vpwait(dev)
     dev_t dev;
{
     register struct vpdevice *vpaddr =
          (struct vpdevice *)vpdinfo[VPUNIT(dev)]->md_addr;
     register struct vp_softc *sc = &vp_softc[VPUNIT(dev)];

     for (;;) {
          if ((sc->sc_state & VPSC_BUSY) == 0 &&
              vpaddr->vp_status & VP_DRDY)
               break;
          sleep((caddr_t)sc, VPPRI);
     }
     return;
}

struct pair {
     char soft;               /* software bit */
     char hard;               /* hardware bit */
} vpbits[] = {
     VPC_RESET,     VP_RESET,
     VPC_CLRCOM,    VP_CLEAR,
     VPC_EOTCOM,    VP_EOT,
     VPC_FFCOM,     VP_FF,
     VPC_TERMCOM,   VP_TERM,
     0,             0,
};

/*
 * vpcmd is designed to be called after vpwait has returned, thus
 * indicating that the hardware is quiet and ready to receive a new command.
 * When it's called, it runs through the possible command bits in
 * sc->sc_state, and, finding one set, issues the corresponding hardware
 * command to the actual device. At the same time it clears the command from
 * sc->sc_state, so that the next time vpcmd is called another
 * command will be issued to the hardware. Note that vpcmd waits a long
 * time, probably too long, for the device to recover before it returns.
 */
vpcmd(dev)
        dev_t;
{
        register struct vp_softc *sc = &vp_softc[VPUNIT(dev)];
        register struct vpdevice *vpaddr =
            (struct vpdevice *)vpdinfo[VPUNIT(dev)]->md_addr;
        register struct pair *bit;

        for (bit = vpbits; bit->soft != 0; bit++) {
                if (sc->sc_state & bit->soft) {
                        vpaddr->vp_cmd = bit->hard;
                        sc->sc_state &= ~bit->soft;
                        DELAY(100);     /* time for DRDY to drop */
                        return;
                }
        }
}

/*ARGSUSED*/
vpioctl(dev, cmd, data, flag)
        dev_t dev;
        int cmd;
        caddr_t data;
        int flag;
{
        register int m;
        register struct mb_device *md = vpdinfo[VPUNIT(dev)];
        register struct vp_softc *sc = &vp_softc[VPUNIT(dev)];
        register struct vpdevice *vpaddr = (struct vpdevice *)md->md_addr;
        int s;

        switch (cmd) {

        case VGETSTATE:
                *(int *)data = sc->sc_state;
                break;

        /*
         * Turn off VPSC_MODE; restrict the user to resetting it and setting
         * VPSC_CMNDS
         */
        case VSETSTATE:
                m = *(int *)data;
                sc->sc_state =
                    (sc->sc_state & ~VPSC_MODE) | (m&(VPSC_MODE|VPSC_CMNDS));
                break;

        default:
                return (ENOTTY);     /* "Not a typewriter" */
        }
```

```
     /*
      * More careful handling to make sure that one command doesn't get issued until the
      * last one has completed. Wait, then post some state information from
      * sc->sc_softc to the hardware, then wait again, then call vpcmd to
      * fire off the next command. And all in a critical section!
      */
     s = splx(pritospl(md->md_intpri));
     vpwait(dev);
     if (sc->sc_state&VPSC_SPP)
          vpaddr->vp_status = VP_SPP|VP_PLOT;
     else if (sc->sc_state&VPSC_PLOT)
          vpaddr->vp_status = VP_PLOT;
     else
          vpaddr->vp_status = 0;
     while (sc->sc_state & VPSC_CMNDS) {
          vpwait(dev);
          vpcmd(dev);
     }
     (void) splx(s);
     return (0);
}

/*
 * This is really a polling interrupt routine. The code at the top that checks
 * the polling chain should really be broken out into a vppoll routine
 * that gets plugged into the mb_device structure. The rest of the code
 * would then be where it properly belongs, in a vpintr routine that can
 * be named in the config file.
 */
vpintr()
{
     register int dev;
     register struct mb_device *md;
     register struct vpdevice *vpaddr;
     register struct vp_softc *sc;
     register int found = 0;

     for (dev = 0; dev < NVP; dev++) {
          if ((md = vpdinfo[dev]) == NULL)
               continue;
          vpaddr = (struct vpdevice *)md->md_addr;

          /*
           * It's not easy to find out if an interrupt has occurred.
           */
          vpaddr->vp_icad0 = VP_ICPOLL;
          DELAY(1);
          if (vpaddr->vp_icad0 & 0x80) {
               found = 1;

               /* Wake up the guilty device */
               DELAY(1);
               vpaddr->vp_icad0 = VP_ICEOI;
```

```
                }

                sc = &vp_softc[dev];

                /* Is there a command currently dispatched and does the hardware
                 * say it's done with it?
                 */
                if ((sc->sc_state&VPSC_BUSY) && (vpaddr->vp_status & VP_DRDY)) {
                        sc->sc_state &= ~VPSC_BUSY;   /* clear busy indicator */
                        if (sc->sc_state & VPSC_SPP) {

                                /*device-specific mode toggle */
                                sc->sc_state &= ~VPSC_SPP;
                                sc->sc_state |=  VPSC_PLOT;
                                vpaddr->vp_status = VP_PLOT;
                        }
                        iodone(sc->sc_bp); /* break wait in physio */
                        sc->sc_bp = NULL;

                        /*
                         * Note that the resources being deallocated here were allocated
                         * in vpstrategy, in the top half of the driver. This is
                         * standard form for DMA drivers.
                         */
                        mbrelse(md->md_hd, &sc->sc_mbinfo);
                }
                wakeup((caddr_t)sc); /* break loops in vpstrategy AND vpwait */
        }
        return (found);
}

/*
 * vptimo is used to repeatedly kickstart the device, which has a tendency
 * to freeze up if left alone too long. It calls vpintr, and then it sets
 * up a timer to call vptimo again (and again, and again...) to make sure
 * that a call to vpintr is always pending. The kernel global hz is set
 * to reflect the clock rate of the system processor chip (it's 50 for a Sun3).
 */
vptimo()
{
        int s;
        register struct mb_device *md = vpdinfo[0];

        s = splx(pritospl(md->md_intpri));
        (void) vpintr();
        (void) splx(s);
        timeout(vptimo, (caddr_t)0, hz);
}
```

E.5. Sun386i Parallel Port Driver

```
/*
 * (ppreg.h) Sun-386i Parallel Port Registers
 * Copyright (c) 1987 by Sun Microsystems, Inc.
 */

/* Register addresses.
 */

ushort ppregs[][NPPREGS] = {
     { 0x378, 0x37a, 0x379 }, /* port 1 regs */
};

/* Printer Control Reg bits */
#define    PC_INTENABLE     0x10  /* +IRQ ENABLE: enable ACK interrupts */
#define    PC_SELECT        0x08  /* +SLCT IN: select printer */
#define    PC_INIT          0x04  /* -INIT: init printer */
#define    PC_LINEFEED      0x02  /* +AUTO FD XT: set auto linefeed */
#define    PC_STROBE        0x01  /* +STROBE: strobe data */

#define    PC_NORM          (PC_INTENABLE|PC_SELECT|PC_INIT)
#define    PC_OFF           (PC_SELECT|PC_INIT)
#define    PC_RESET         0

/* Printer Status Reg bits */
#define    PS_READY         0x80  /* -BUSY: printer not busy */
#define    PS_NOTACK        0x40  /* -ACK: ACK state */
#define    PS_NOPAPER       0x20  /* +PE: printer out of paper */
#define    PS_SELECT        0x10  /* +SLCT: printer is selected */
#define    PS_NOERROR       0x08  /* -ERROR: printer error condition */

#define    PSREADY(s)       ((s)&PS_READY)
#define    PSSELECT(s)      ((s)&PS_SELECT)
#define    PSNOPAPER(s)     ((s)&PS_NOPAPER)
#define    PSERROR(s)       (((s)&PS_NOERROR) == 0)
```

```
/*
 * Parallel Port (printer) driver.
 * Copyright (c) 1987 by Sun Microsystems, Inc.
 */

#include "pp.h"
#if NPP > 0

#include <sys/param.h>
#include <sys/buf.h>
#include <sys/uio.h>
#include <sys/errno.h>
#include <sys/file.h>
#include <sundev/mbvar.h>

/*
 * Buffers for use by physio().
 */
struct buf ppbuf[NPP];
#define    PPBUFSIZ   64     /* Size of buffer written to printer */

/*
 * Software state structure, one for each printer
 */
struct ppstate {
  int      pp_flags;              /* Printer state: */
#define    PP_OPEN    0x01        /*     Currently open */
#define    PP_WANT    0x02        /*     Someone waiting for printer */
#define    PP_TIMER   0x08        /*     Watchdog timer is running */
#define    PP_BUSY    0x10        /*     I/O in progress */
  u_char   pp_timer;              /* For detecting timeout situations*/
  u_char   pp_lostintr;           /* For tracking lost interrupts*/
  u_char   pp_notready;           /* Printer not ready (no paper, etc.) */
  int      pp_unit;               /* Unit number*/
  struct   mb_device *pp_md;      /* Pointer to mb info */
  struct   buf   *pp_bp;          /* Pointer to current 'buf' */
  char     pp_buf[PPBUFSIZ];      /* Buffer */
  char     *pp_cp;                /* Current byte in current buffer */
  int      pp_count;              /* Number of bytes left to print */
  u_short pp_regbase;             /* Device register base in i/o space */
} ppstate[NPP];

#define    PPREG_DATA     (pp->pp_regbase)
#define    PPREG_CTRL     (pp->pp_regbase + 2)
#define    PPREG_STAT     (pp->pp_regbase + 1)


#define    PPUNIT(dev)    (minor(dev))
#define    PPPRI          (PZERO + 1)     /* Sleeps are interruptable */

extern int hz;
#define    PPWATCHDOG     3      /* Watchdog interval: see 'pptimeout()' */
#define    PPTICKS        (30/PPWATCHDOG + 1)
```

```
#define   PPMSGTICKS     (180/PPWATCHDOG)

#ifdef DEBUG
/*
 * Debugging stuff.
 */
#define DBINIT      0x0001
#define DBIO        0x0002
#define DBOPEN      0x0004
#define DBCLOSE     0x0008
#define DBSTRAT     0x0010
#define DBSTART     0x0020
#define DBTMOUT     0x0040
#define DBINTR      0x0080

int ppdebug = 0xffff;
#define ppprint(flg,x)   (((flg)&ppdebug) ? printf x : 0)

#else
#define ppprint(flg,x)
#endif DEBUG

int ppprobe(), ppattach(), ppintr(), pptimeout();

struct mb_driver ppdriver = {
     ppprobe, 0, ppattach, 0, 0, ppintr, 0, "pp", 0, 0, 0, 0,
};

/*ARGSUSED*/
ppprobe(reg, unit)
     caddr_t reg;
     int unit;
{
     ppprint(DBINIT, ("ppprobe\n"));

     if (unit >= NPP)
          panic("pp: too many units");

     ppstate[unit].pp_regbase = (u_short)reg;
     return(1);
}

ppattach(md)
     register struct mb_device *md;
{
     register struct ppstate *pp;

     ppprint(DBINIT, ("ppattach\n"));

     pp = &ppstate[md->md_unit];
     pp->pp_md = md;

     /* Initialize printer.
```

```
      * Holding PC_INIT low for 50 usecs does the trick.
      */
     outb(PPREG_CTRL, PC_RESET);
     DELAY(50);
     outb(PPREG_CTRL, PC_OFF);
     DELAY(10);
}

ppopen(dev, flags)
     dev_t dev;
     int flags;
{
     register struct ppstate *pp = &ppstate[PPUNIT(dev)];
     int status;

     ppprint(DBOPEN, ("ppopen: unit %d\n", PPUNIT(dev)));

     if (PPUNIT(dev) >= NPP  ||  pp->pp_md->md_alive == 0)
          return(ENXIO);
     if (flags & FREAD)           /* Can't read a write-only device */
          return(ENODEV);

     pp->pp_unit = PPUNIT(dev);

     while (pp->pp_flags & PP_OPEN) {   /* Enforce exclusive access */
          ppprint(DBOPEN, ("ppopen: in use - waiting...\n"));
          if (flags & FNDELAY)
               return(EBUSY);
          pp->pp_flags |= PP_WANT;
          if (sleep((caddr_t)&pp->pp_flags, PPPRI|PCATCH)) {
               return(EINTR);
          }
     }

     status = inb(PPREG_STAT);
     if (PSNOPAPER(status)  ||  ! PSSELECT(status)  ||  PSERROR(status)) {
          if (PSNOPAPER(status))
               uprintf("pp%d: printer out of paper\n", pp->pp_unit);
          else
               uprintf("pp%d: printer not ready\n", pp->pp_unit);
          (void)wakeup((caddr_t)&pp->pp_flags);
          pp->pp_flags = 0;
          return(EIO);
     }

     outb(PPREG_CTRL, PC_NORM);    /* Enable interrupts */

     if ((pp->pp_flags & PP_TIMER)  ==  0) {
          /*
           * Kick off watchdog timer.
           */
          timeout(pptimeout, (caddr_t)pp, PPWATCHDOG*hz);
          pp->pp_timer = 0;
          pp->pp_flags |= PP_TIMER;
     }

     pp->pp_flags |= PP_OPEN;
     return(0);
}

/*
 * ppclose:
 *      Close the printer device.
 *      Turn off interrupts.
 *      Wake up anyone waiting to open the printer.
 */
ppclose(dev)
     dev_t dev;
{
     register struct ppstate *pp = &ppstate[PPUNIT(dev)];

     ppprint(DBCLOSE, ("ppclose: unit %d\n", PPUNIT(dev)));

     outb(PPREG_CTRL, PC_OFF);          /* Disable interrupts */

     if (pp->pp_flags & PP_WANT)
          wakeup((caddr_t)&pp->pp_flags);
     pp->pp_flags = 0;
}


ppwrite(dev, uio)
     dev_t dev;
     struct uio *uio;
{
     int ppminphys(), ppstrategy();

     ppprint(DBIO, ("ppwrite\n"));

     return(physio(ppstrategy, &ppbuf[PPUNIT(dev)], dev, B_WRITE,
         ppminphys, uio));
}

/*
 * ppstrategy:
 */
ppstrategy(bp)
     register struct buf *bp;
{
     register struct ppstate *pp = &ppstate[PPUNIT(bp->b_dev)];

     ppprint(DBSTRAT|DBIO, ("ppstrategy\n"));

     pp->pp_bp = bp;
     pp->pp_count = bp->b_bcount;
     pp->pp_cp = pp->pp_buf;
	if (copyin(bp->b_un.b_addr, pp->pp_buf, bp->b_bcount)) {
		bp->b_flags |= B_ERROR;
		bp->b_error = EFAULT;
		ppiodone(pp);
		return;
	}

	pp->pp_flags |= PP_BUSY;
	pp->pp_timer = PPTICKS;		/* Set timer */
	pp->pp_lostintr = 0;		/* Reset "lost interrupt" counter */
	pp->pp_notready = 0;		/* Reset "notready" counter */
	ppintr();
	ppiowait(pp, bp);
	pp->pp_timer = 0;		/* Turn off timer */

	ppprint(DBSTRAT, ("ppstrategy: ***done\n"));
}

ppminphys(bp)
	register struct buf *bp;
{
	if (bp->b_bcount > PPBUFSIZ)
		bp->b_bcount = PPBUFSIZ;
}

/*
 * ppintr:
 *	Handle 'ack' interrupts from printer.
 */
ppintr()
{
	register struct ppstate *pp;
	int status;	/* printer status */
	int s;

	ppprint(DBINTR, ("ppintr\n"));

	pp = &ppstate[0];		/* XXX - only works for unit #0 */

	s = splx(pritospl(pp->pp_md->md_intpri));

	status = inb(PPREG_STAT);
	ppprint(DBINTR, ("ppintr: status = 0x%x\n", status));

	/* Were we expecting an interrupt? */
	if ( ! (pp->pp_flags & PP_BUSY)) {
		ppprint(DBINTR, ("ppintr: unsolicited interrupt\n"));
		splx(s);
		return;
	}

	if (pp->pp_count > 0) {
```

```
          /* AT Tech Ref says data must be in data reg at least
           * 0.5 usec before and after we strobe, and strobe must
           * last at least 0.5 usec.
           */
          outb(PPREG_DATA, *pp->pp_cp);
          pp->pp_cp++;
          pp->pp_count--;
          DELAY(1);
          outb(PPREG_CTRL, PC_NORM|PC_STROBE);
          DELAY(1);
          outb(PPREG_CTRL, PC_NORM);
     }

     else
          ppiodone(pp);

     splx(s);
}

/*
 * pptimeout:
 *      Check occasionally for lost interrupts or
 *      printer errors (no paper, printer off line, etc.).
 */
pptimeout(arg)
     caddr_t arg;
{
     register struct ppstate *pp = (struct ppstate *)arg;
     int  status;    /* Printer status */
     int  error = 0;
     int  s;

     ppprint(DBTMOUT, ("pptimeout\n"));

     s = splx(pritospl(pp->pp_md->md_intpri));


     /* If we're not currently doing anything, we can go away. */
     if ((pp->pp_flags & PP_OPEN)  ==  0) {   /* Not open */
          splx(s);
          return;
     } else if (pp->pp_timer <= 0) {          /* Not currently active */
          timeout(pptimeout, (caddr_t)pp, PPWATCHDOG*hz);
          splx(s);
          return;
     }

     status = inb(PPREG_STAT);

     /* Check for printer errors. */
     if (PSNOPAPER(status)) {
          if ((pp->pp_notready++ % PPMSGTICKS)  ==  0)
               uprintf("pp%d: printer out of paper\n", pp->pp_unit);
```

```
    } else if ( ! PSSELECT(status)  ||  PSERROR(status)) {
        if ((pp->pp_notready++ % PPMSGTICKS)  ==  0)
            uprintf("pp%d: printer not ready\n", pp->pp_unit);
    } else if (--pp->pp_timer == 0) {
        /* Timer has expired - see what's wrong. */
        ppprint(DBTMOUT, ("pptimeout: status = 0x%x\n", status));

        if (PSREADY(status)) {
            /*
             * We must have dropped an interrupt.
             * If this is the first one we've dropped, assume
             * it's a fluke and carry on.  Otherwise, give up.
             */
            if (pp->pp_lostintr++ == 0) {
                ppprint(DBTMOUT, ("pptimeout: dropped intr\n"));
                pp->pp_timer = PPTICKS;   /* Reset timer */
                ppintr();
            } else {
                printf("pp%d: not getting interrupts\n",
                    pp->pp_unit);
                error = 1;
            }
        } else {
            /* Printer is hung */
            error = 1;
        }
    }

    if ( ! error) {
        timeout(pptimeout, (caddr_t)pp, PPWATCHDOG*hz);
    } else {
        pp->pp_bp->b_flags |= B_ERROR;
        ppiodone(pp);
        pp->pp_flags &= ~PP_TIMER;
    }

    splx(s);
}


/*ARGSUSED*/
ppioctl(dev, cmd, data, flag)
    dev_t dev;
    int cmd;
    caddr_t data;
    int flag;
{
    return(ENOTTY);
}

/*
 * ppiowait:
 *      Private version of 'biowait()'.
 */
ppiowait(pp, bp)
     struct ppstate *pp;
     register struct buf *bp;
{
     int s;

     s = splx(pritospl(pp->pp_md->md_intpri));
     while ( ! (bp->b_flags&B_DONE)) {
          if (sleep((caddr_t)bp, PPPRI|PCATCH)) {
               bp->b_flags |= (B_ERROR|B_DONE);
               bp->b_error = EINTR;
          }
     }

     splx(s);
}

/*
 * ppiodone:
 *     Private version of 'biodone()'.
 */
ppiodone(pp)
     register struct ppstate *pp;
{
     register struct buf *bp = pp->pp_bp;

     bp->b_flags |= B_DONE;
     wakeup((caddr_t)bp);

     pp->pp_flags &= ~PP_BUSY;
}

#endif NPP
```

Index

8

- 80386, 25

A

- accessing the datagram provider, STREAMS, 228
- `adb` command, 99
- adding STREAMS modules, 242
- addresses
  - convenient testing, 78
  - DVMA virtual, 19
  - finding physical, 79
  - kernel space, 63
  - mapping of, 76
  - mapping Sun-2, 77, 81
  - mapping Sun-3, 77, 84
  - mapping Sun-4, 78, 84
  - mapping Sun386i, 78
  - selection of virtual, 76
  - space terminology, 7
  - user space, 63
  - virtual space warning, 6
  - virtual to physical mapping, 79
- advanced operations, STREAMS, 210
- advanced topics, STREAMS, 299
- assert mechanism, 104
- asynchronous
  - I/O, STREAMS, 213
  - tracing, 101
- asynchronous notification, 134
- ATbus Machines, 25
- `attach()` routine, 67, 117
- autoconfiguration, 61
  - and initialization, 48
  - related declarations, 55
  - Skeleton example, 115

B

- `bappend()`, 250
- basic operations, STREAMS, 204
- `bdevsw`, 42
  - definition, 139
- block driver mechanisms, 5
- bottom half of driver, 63, 64
- 4.2BSD, 48
- buffer allocation priority, STREAMS, 328
- building a kernel, 139
- building a multiplexor, STREAMS, 216
- byte order, 28

C

- CANONPROC, 186
- `cdevsw`, 42, 142
  - definition, 139
- character driver overview, 61
- clone open, STREAMS, 214
- cloning, STREAMS, 269
- `close()` routine, 117
- closing STREAMS, 242, 277
- computer architecture, 13
- `config` command, 140
- configuration, **139**, 140
  - autoconfiguration, 140
  - `conf.c`, 142
  - config file, 141
  - configuration makefile, 140
  - device installation, 140
  - dual address-space devices, 145
  - example, 141
  - file, 141
  - GENERIC file, 23
  - MAKEDEV shell script, 143
  - `mknod` command, 144
- context registers, 79
- controllers, 49
- CPU PROM monitor, 75 *thru* 90
  - warning, 90
- CPU state, 80
- critical sections, 64

D

- data structures
  - kernel, 48
  - STREAMS, 339
- data, routing multiplexed, STREAMS, 222
- datagram
  - receiving a datagram, STREAMS, 232
  - sending a datagram, STREAMS, 231
  - service interface, STREAMS, 226
- Debugging Techniques, 97
- definition of "Stream" in STREAMS context, 202
- `/dev` directory, 42
- development facilities, STREAMS, 203

- device
  - as special files, 42
  - block devices, 42
  - character devices, 42
  - classes, 42
  - devices and controllers, 49
  - independence, 3
  - initial checkout, 88
  - installation, 144
  - major numbers, 42
  - major types, 4
  - memory-mapped installation, 90
  - minor numbers, 42
  - names, 42
  - number macros, 72
  - numbers, 42
  - peculiarities, 28
  - preassigned devices, 46
  - slave vrs free devices, 49
  - testing, 76
  - tty-like devices, 46
  - virtual-memory, 92
  - warnings, 28
- Device Drivers
  - introduction, 3
  - kernel space, 63
  - regular drivers, 9
  - types of devices, 4
- dismantling a multiplexor, STREAMS, 221
- DMA
  - devices, 33, 126
  - Multibus, 126
  - Skeleton Board DVMA, 127
  - VMEbus, 126
- `dmesg` command — See system messages, 101
- DOS and SunOS drivers, 27
- driver
  - kernel interaction, 41
  - kernel interface, 56
  - overview, 61
  - source code, 143
  - STREAMS close, 269
  - STREAMS declarations, 262
  - STREAMS development facilities, 238
  - STREAMS environment, 246
  - STREAMS flow control, 261
  - STREAMS flush handling, 266
  - STREAMS interrupt, 266
  - STREAMS ioctls, 267
  - STREAMS open, 264
  - STREAMS processing procedures, 265
  - STREAMS programming, **237**, 262
  - user processes, 41
- driver example, 111
- driver listing
  - color graphics driver, 398
  - skeleton driver, 390
  - Sky floating-point driver, 415
  - Sun386i parallel port driver, 435
  - Versatec interface driver, 423
- driver routines, **351**
  - `xxattach()`, 352, 357
  - `xxclose()`, 352
- driver routines, *continued*
  - `xxintr()`, 352
  - `xxioctl()`, 353
  - `xxminphys()`, 355
  - `xxmmap()`, 355
  - `xxopen()`, 356
  - `xxpoll()`, 356
  - `xxprobe()`, 357
  - `xxread()`, 358
  - `xxselect()`, 358
  - `xxstrategy()`, 359
  - `xxwrite()`, 359
- dual address-space devices, 145
- DVMA, **33**
  - DVMA hardware, 34
  - DVMA space, 35
  - DVMA variable, 36
  - no user-level DVMA, 36
  - `rmalloc()`, 35
  - Sun Main Bus DVMA, 34

E

- error
  - logging, 104
  - numbers, 351
  - recovery, 103
  - returns, 103
  - signals, 104
  - STREAMS error messages, 310
  - STREAMS logging, 193
- example
  - configuration, 141
  - mapping without drivers, 94, 96
  - `mmap()`, 91
  - PTE calculations, 87
  - ramdisk pseudodevice, 151
  - STREAMS asynchronous protocol, 174
  - STREAMS message use, 177
  - STREAMS multiplexing, 281
  - STREAMS multiplexor configuration, 287
  - STREAMS, big example, 269
- example driver, 111
- external STREAMS variables, 243

F

- filesystems, 5
- filter module declarations, STREAMS, 249
- flow control, STREAMS, 188, 255, 342
- frame buffers, 90
  - mapping without drivers, 93
- freeing messages, 243

H

- hardware peculiarities, 28
- `hat_getkpfnum()`, 91
- heterogeneous networks, 139

I

- I/O
  - and signals, 130
  - asynchronous, 130
  - asynchronous notification, 130

- I/O, *continued*
  - non-blocking, 130
  - paths, 43
  - STREAMS advanced, 210
  - STREAMS asynchronous, 213
  - STREAMS polling, 210
- initial
  - checkout, 88
  - declarations, 61
  - device tests, 89
- inserting modules, STREAMS, 206
- installation of device, 144
- Intel 80386, 25
- interface, message, STREAMS, 224
- interrupt
  - context, 63
  - levels, 65
  - number setting, 67
  - related problems, 33
  - routines, 54, 62
  - vector assignments, 24
- interrupts, 64
  - polling, 66
  - vectored, 66
- `intr()` routine, 62, 124
- `ioctl()`
  - macros, 354
  - routine, 62, 126, 134

K

- `kadb` — the kernel debugger, 102
  - abort to monitor, 102
  - and virtual spaces, 102
  - limitation, 102
- kernel
  - buffer cache, 5
  - config file, 141
  - configuration, 139
  - data structures, 48
  - interface, 48
  - interface points, 115
  - kernel/driver interface, 56
  - memory context, 41
  - panics, 104
  - run-time data structures, 47
  - space, 63
  - STREAMS functions, 180
  - STREAMS structures, 317
- KERNELBASE, 63

L

- limitations of this manual, 5
- line disciplines, 312
- Loadable Drivers, 27, 105
  - Adding, 146
  - Removing, 146
- `log` command, 193
- loop-around driver, STREAMS, 270

M

- Main Bus, 48
  - resource management, 48
- `major()` macro, 72
- MAKEDEV shell script, 143
- `makedev()` macro, 72
- manual overview, 8
- MAP_FIXED, 386
- MAP_PRIVATE, 386
- MAP_RENAME, 386
- MAP_SHARED, 386
- MAP_TYPE, 386
- mapping without drivers, examples, 94, 96
- `mb_ctlr` structure, 50, 67
- `mb_device` structure, 51, 67
- `mb_driver` structure, 52
- `mb_hd` structure, 49
- `mbglue.s`, 140
- `mbvar` structures, 48
- `mc_addr` field, 50
- `mc_alive` field, 51
- `mc_ctlr` field, 50
- `mc_dmachan` field, 50
- `mc_intpri` field, 50
- `mc_intr` field, 50
- `mc_mbinfo` field, 51
- `mc_space` field, 50
- MC680X0, 13
- `md_addr` field, 51
- `md_alive` field, 52
- `md_dmachan` field, 51
- `md_driver` field, 51
- `md_flags` field, 52
- `md_intpri` field, 51
- `md_intr` field, 52
- `md_slave` field, 51
- `md_unit` field, 51
- `mdr_attach` field, 53
- `XDR_BIODMA`, 55
- `mdr_cinfo` field, 54
- `mdr_cname` field, 54
- `mdr_dinfo` field, 54
- `XDR_DMA`, 55
- `mdr_dname` field, 54
- `mdr_done` field, 53
- `mdr_flags` field, 55
- `mdr_go` field, 53
- `mdr_intr` field, 53
- `mdr_link` field, 55
- `XDR_OBIO`, 55
- `mdr_probe` field, 53
- `mdr_size` field, 54
- `mdr_slave` field, 53
- `XDR_SWAB`, 55
- `XDR_XCLU`, 55
- memory contexts, 79
- Memory Management Unit, 15
- memory mapping, 13, 90

memory-mapped
  device installation options, 90
  devices, 63, 90
  drivers, 90
message allocation, STREAMS, 182, 251
message blocks, STREAMS, 181
message form and linkage, STREAMS, 247
message format, STREAMS, 247
message generation, STREAMS, 249
message handling, STREAMS, 223 *thru* 234
message interface, STREAMS, 224
message priority, STREAMS, 254
message queues, STREAMS, 180, 253, 254
message reception, STREAMS, 249
message structures, STREAMS, 318
message types, STREAMS, 176, 320
`minor()` macro, 72
`minphys()` routine, 120
`mknod` command, 144
`mmap()`, **90**, 62
  direct opening of devices, 95
  `mmap()`, 91
  without drivers, 92
MMU
  setting the, 75
  Sun-2, 81
  Sun-3, 84
  Sun-4, 84
modularity STREAMS, 162
module and driver control, STREAMS, 207
module configuration, STREAMS, 308
module declarations, STREAMS, 243
module environment, 246
module ioctls, STREAMS, 267
module procedures, 245
Module Programming, **237**
module reusability, STREAMS, 167
monitor, 75 *thru* 90
  warning, 90
Multibus, **13**
  3.0 changes, 17
  adapter, 24
  adapter warning, 31
  byte-ordering issues, 28
  device peculiarities, 28
  DMA, 126
  I/O mapped devices, 14
  I/O space, 13
  I/O Space allocation, 17
  memory allocation, 16
  memory mapped devices, 14
  memory space, 13
  memory types, 14
  MMU, 15
  multibus resource management, 71
  other peculiarities, 30
  Sun-2 Multibus, 15
  Sun-2 Multibus memory map, 16
multiple address-space devices, 145
multiplexed data, routing, STREAMS, 222
multiplexed Streams, STREAMS, 214 *thru* 222
multiplexing driver, STREAMS, 284
multiplexing,
  STREAMS", 278

N

`noprintf` variable, 101

O

`open()` routine, 117
opening a Stream, 241
overview of STREAMS drivers, 259

P

P2 bus, 31
Page Map Entry Groups, PMEGs, 80
page maps, 80
Page Table Entries, PTEs, 80
pixrects, 90
PMEGS, 80
`poll()` routine, 62, 67, 124
polling
  chain, 54
  interrupts, 66
  restrictions on, 66
  STREAMS I/O, 210
`printf()`
  event triggered, 100
  restrictions on, 99
  usage hints, 100
  with debuggers, 99
priority messages, STREAMS, 325
`probe()` routine, 115
`proc` structure, 57
processes, 80
processor priority, 64
  raising and lowering, 70
processor state, 80
PROM monitor, 75 *thru* 90
  warning, 90
`PROT_EXECUTE`, 386
`PROT_READ`, 386
`PROT_WRITE`, 386
protocol portability, STREAMS, 165
protocol substitution, STREAMS, 166
pseudo devices, **151**
  ramdisk example, 151
PTE, 80
  calculations, 87
  Sun-2 masks, 82
  Sun-2 PTE, 82
  Sun-3 masks, 86
  Sun-4 masks, 86
  templates, 82, 86
put procedure rules, STREAMS, 341
put procedure, STREAMS, 251
put procedures, STREAMS, 183

Q

- QUEUE data structures, 240
- queue priority, STREAMS, 254
- `queue_t`, 253

R

- ramdisk
  - driver, 152
  - installation, 153
  - source code, 152
  - test program, 156
- `read()` routine, 62, 119
- receiving messages, STREAMS, 176
- receiving, a datagram, STREAMS, 232
- register
  - peculiarities, 28
  - sequencing logic, 32
  - warnings, 28
- removing STREAMS modules, 242
- routing multiplexed data, STREAMS, 222
- run-time data structures, 47

S

- sample listings, 389
- segment maps, 80
- Select Routines, 131
- `select()`, 62, 131
  - and `ioctl()`, 134
  - interrupt time, 133
- `select()` routine, 130
- `selwait()`, 133
- semaphores, 379
- sending messages, STREAMS, 176
- service functions, 69
  - change processor priorities, 70
  - data-transfer functions, 71
  - multibus resource management, 71
  - `printf()`, 72
  - sleep and wakeup, 69
  - timeout, 69
  - untimeout, 69
- service interface messages, STREAMS, 223
- service interface, datagram, STREAMS, 226
- service interface, STREAMS, 294
- service procedure rules, STREAMS, 341
- service procedures, STREAMS, 183, 253, 254, 276
- service, closing a, STREAMS, 231
- Skeleton driver, 111
- Skeleton driver declarations, 114
- sleep and wakeup mechanism, 69
- `sleep()` system call, 64, 65
- software devices, ramdisk example, 151
- software priorities, 70
- SPARC
  - and MC680X0, 29
  - and Multibus, 28
  - peculiarities, 31
- `start()` routine, 62, 122
- `strace` command, 193
- `strategy()` routine, 121
- Stream construction, 239
- Stream end, 170
- Stream head, 169
- Stream head messages, 276
- STREAMS
  - accessible functions, 340
  - accessible symbols, 340
  - adding modules, 242
  - advanced operations, 210
  - advanced topics, 299
  - advanced topics, flow control, 301
  - advanced topics, read options, 303
  - advanced topics, recovering from no buffers, 299
  - advanced topics, signals, 302
  - advanced topics, Stream head processing, 303
  - advanced topics, write offset, 303
  - advanced view, 168
  - asynchronous protocol example, 174
  - `bappend()`, 250
  - basic operations, 204
  - basic view, 162
  - benefits, 165
  - big example, 269
  - buffer allocation priority, 328
  - building a multiplexor, 216
  - building a Stream, 171
  - CANONPROC, 186
  - clone open, 214
  - cloning, 269
  - close, 277
  - closing, 242
  - closing a service, 231
  - configuring drivers, 307
  - creating service interfaces, 165
  - data structures, 339
  - datagram provider access, 228
  - datagram service interface, 226
  - datagram, receiving a, 232
  - definition of "Stream", 202
  - design and system calls, 339
  - design guidelines, 338
  - development facilities, 203
  - device driver Streams, 261
  - dismantling a multiplexor, 221
  - driver `cdevsw` interface, 307
  - driver close, 269
  - driver declarations, 262
  - Driver development facilities, 338
  - driver environment, 246
  - driver flow control, 261
  - driver flush handling, 266
  - driver interrupt, 266
  - driver ioctls, 267
  - driver open, 264
  - driver processing, 185
  - driver programming, 262
  - drivers, 195
  - environment, 195
  - error and trace logging, 193
  - error messages, 310
  - expanded Streams, 172
  - external variables, 243
  - filter module declarations, 249

STREAMS, *continued*
- flow control, 188, 255, 342
- freeing messages, 243
- functional parts, 163
- functions, accessible, 340
- general design rules, 338
- glossary, 196, 343
- header files, 340
- I/O, advanced, 210
- I/O, asynchronous, 213
- I/O, polling, 210
- inserting modules, 206
- Internet multiplexing, 190
- Internet multiplexor after connecting, 283
- Internet multiplexor before connecting, 282
- introduction, 161
- ioctls, 267
- kernel level functions, 180
- kernel processing, 184
- kernel structures, 317
- kernel structures, `iocblk`, 319
- kernel structures, `linkblk`, 319
- kernel structures, `QUEUE`, 317
- kernel structures, `streamtab`, 317
- line disciplines, 312
- loop-around driver, 270
- `M_PCPROTO` messages, 321
- `M_PROTO` messages, 321
- manipulating modules, 165
- manipulating STREAMS modules, 165
- manual pages, 164
- mechanism, 238
- message allocation, 182, 251
- message blocks, 181
- message form and linkage, 247
- message format, 247
- message generation, 249
- message handling, 223 *thru* 234
- message interface, 224
- message priority, 254
- message queue priority, 187
- message queues, 180, 253, 254
- message reception, 249
- message structures, 318
- message types, 176, 320
- message types, ordinary, 320
- message use in example, 177
- modularity, 162
- module and driver control, 207
- module configuration, 308
- module declarations, 243
- module environment, 246
- module ioctls, 267
- module procedures, 245
- module reusability, 167
- modules, 169, 196
- monitoring, 192
- multiplexed Streams, 214 *thru* 222
- multiplexing, 190, 278
- multiplexing configurations, 278
- multiplexing driver, 284
- multiplexing, connecting lower Streams, 279
- multiplexing, disconnecting lower Streams, 281
- multiplexing, example, 281

STREAMS, *continued*
- multiplexor configuration, example, 287
- multiplexor, lower QUEUE write, 290
- multiplexor, lower read put, 292
- multiplexor, upper write put, 287
- opening a Stream, 241
- overview of drivers, 259
- portability, 312
- primer, 161
- priority messages, 325
- protocol migration, 166
- protocol portability, 165
- protocol substitution, 166
- pushable modules, 172
- put and service procedures, 183
- put procedure, 251
- put procedure rules, 341
- QUEUE data structures, 240
- queue priority, 254
- `queue_t`, 253
- receiving a datagram, 232
- removing modules, 242
- routing multiplexed data, 222
- sending a datagram, 231
- sending and receiving messages, 176
- service interface, 294
- service interface messages, 223
- service interface procedure, 297
- service interface, declarations, 295
- service interface, messages, 294
- service procedure rules, 341
- service procedures, 253, 254, 276
- single I/O pathway, 202
- standard SunOS modules, 311
- Stream construction, 239
- Stream end, 170
- Stream head, 169
- Stream head messages, 276
- `streamtab`, 308
- SunOS, 311
- SunOS extension, 312
- SunOS modules, 311
- SunOS STREAMS Topics, 307
- supplementary material, 317
- symbols, accessible, 340
- system calls, 163
- system error messages, 310
- tunable parameters, 309
- user line disciplines, 312
- utilities, 327
- write put procedure, 273
- write side processing, 186
- X.25 multiplexing, 191

STREAMS application programming, 201 *thru* 234

STREAMS Drivers, 8

STREAMS service interfaces, 165

STREAMS System Calls, 173

STREAMS utilities
- `adjmsg()`, 329
- `allocb()`, 329
- `backq()`, 329
- `bufcall()`, 330
- `canput()`, 330

STREAMS utilities, *continued*
  - `copyb()`, 330
  - `copymsg()`, 331
  - `dupb()`, 331
  - `dupmsg()`, 331
  - `enableok()`, 332
  - `flushq()`, 332
  - `freeb()`, 332
  - `freemsg()`, 332
  - `insq()`, 333
  - `linkb()`, 333
  - `msgdsize()`, 333
  - `noenable()`, 333
  - `OTHERQ()`, 334
  - `pullupmsg()`, 334
  - `putbq()`, 334
  - `putctl()`, 334
  - `putctl1()`, 335
  - `putnext()`, 335
  - `putq()`, 335
  - `qenable()`, 336
  - `qreply()`, 336
  - `qsize()`, 336
  - `RD()`, 336
  - `rmvb()`, 336
  - `rmvq()`, 337
  - `setq()`, 332
  - `splstr()`, 337
  - `strlog()`, 337
  - `sx()`, 331
  - `testb()`, 337
  - `unlinkb()`, 338
  - `WB()`, 338
  - `x()`, 330

`strerr` command, 193

`strlog()`, 193

Sun-4 Peculiarities, 31

Sun386i
  - address mapping', 78
  - DMA, 71
  - DMA Channels, 27
  - DMA on ATbus machines, 36
  - `dma_done()`, 364
  - `dma_setup()`, 364
  - DOS driver, 27
  - `inb()`, 368
  - interrupts, 26
  - loadable drivers, 27, 105, 146
  - no DVMA, 33
  - no vectored interrupts, 51
  - `outb()`, 373

SunOS source license, 5

support routines
  - `btodb()`, 363
  - `CDELAY()`, 364
  - `copyin()`, 363
  - `copyout()`, 363
  - `DELAY()`, 364
  - `dma_done()`, 364
  - `dma_setup()`, 364
  - `gsignal()`, 368
  - `hat_getkpfnum()`, 368
  - `inb()`, 368

support routines, *continued*
  - `iodone()`, 369
  - `iowait()`, 369
  - `kmem_alloc()`, 369
  - `kmem_free()`, 369
  - `log()`, 370
  - `mapin()`, 370
  - `mapout()`, 372
  - `MBI_ADDR()`, 370
  - `mbrelse()`, 372
  - `mbsetup()`, 372
  - `outb()`, 373
  - `panic()`, 373
  - `peek()`, 373
  - `peekc()`, 373
  - `peekl()`, 373
  - `physio()`, 373
  - `poke()`, 375
  - `pokec()`, 375
  - `pokel()`, 375
  - `printf()`, 376
  - `pritospl()`, 376
  - `psignal()`, 377
  - `rmalloc()`, 377
  - `rmfree()`, 377
  - `selwait()`, 378
  - `selwakeup()`, 378
  - `sleep()`, 378
  - `spl`*n*`()`, 379
  - `splx()`, 379
  - `suser()`, 380
  - `swab()`, 380
  - `timeout()`, 380
  - `uiomove()`, 380
  - `untimeout()`, 381
  - `uprintf()`, 381
  - `ureadc()`, 381
  - `uwritec()`, 381
  - `wakeup()`, 382

system calls, 42, 63

system calls, STREAMS, 163

system configuration, 139

System DVMA, 35

system memory devices, 92

system reset, 75

system upgrades, 105

System V compatibility, 6

System V differences, 48

T

timeout mechanisms, 69

timing problems, 33

top half of driver, 63

trace logging, STREAMS, 193

tracing, 101

tunable parameters, STREAMS, 309

U

u structure, 56

`uio` structure, 120

upgrades, 105

user context, 63

user space, 63
`user` structure, 56
user-level routines
    `free()`, 385
    `getpagesize()`, 385
    `mmap()`, 385
    `munmap()`, 386
utilities, STREAMS, 327

V

`v_func` field, 50
`v_vec` field, 50
`v_vptr` field, 50
vector numbers, 67
vectored interrupts, 66
virtual memory devices, 92
virtual to physical mapping, 79
VMEbus, **18**
    16-bit allocation, 22
    24-bit allocation, 23
    32-bit allocation, 23
    allocation of VMEbus memory, 22
    device address assignments, 24
    DMA, 126
    generic, 20
    Multibus Adapter, 24
    Sun-2 VMEbus, 18
    Sun-2 VMEbus address spaces, 18
    Sun-2 VMEbus memory types, 18
    Sun-3 address spaces, 20
    Sun-3 VMEbus, 21
    Sun-3 VMEbus address types, 20
    Sun-4 address spaces, 20
    Sun-4 VMEbus, 22
    Sun-4 VMEbus address types, 20
VMEbus machines, 18

W

write put procedure, STREAMS, 273
`write()` routine, 62, 119

Notes

Notes

Notes

Notes

Notes

Notes