HP 9000 Networking

STREAMS/UX for the HP 9000 Reference Manual

HP Part No. J2237-90005
Printed in U.S.A.
E0195

Edition 2
© Copyright 1995, Hewlett-Packard Company.

Legal Notices

The information in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be held liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Warranty. A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

Restricted Rights Legend. Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 for DOD agencies, and subparagraphs (c) (1) and (c) (2) of the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 for other agencies.

HEWLETT-PACKARD COMPANY
3000 Hanover Street
Palo Alto, California 94304 U.S.A.

Use of this manual and flexible disk(s) or tape cartridge(s) supplied for this pack is restricted to this product only. Additional copies of the programs may be made for security and back-up purposes only. Resale of the programs in their present form or with alterations, is expressly prohibited.

Copyright Notices. ©copyright 1983-95 Hewlett-Packard Company, all rights reserved.

Reproduction, adaptation, or translation of this document without prior written permission is prohibited, except as allowed under the copyright laws.

©copyright 1979, 1980, 1983, 1985-93 Regents of the University of California

This software is based in part on the Fourth Berkeley Software Distribution

under license from the Regents of the University of California.

©copyright 1980, 1984, 1986 Novell, Inc.
©copyright 1986-1992 Sun Microsystems, Inc.
©copyright 1985-86, 1988 Massachusetts Institute of Technology.
©copyright 1989-93 The Open Software Foundation, Inc.
©copyright 1986 Digital Equipment Corporation.
©copyright 1990 Motorola, Inc.
©copyright 1990, 1991, 1992 Cornell University
©copyright 1989-1991 The University of Maryland
©copyright 1988 Carnegie Mellon University

Trademark Notices UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

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

MS-DOS and Microsoft are U.S. registered trademarks of Microsoft Corporation.

OSF/Motif is a trademark of the Open Software Foundation, Inc. in the U.S. and other countries.

Printing History

The manual printing date and part number indicate its current edition. The printing date will change when a new edition is printed. Minor changes may be made at reprint without changing the printing date. The manual part number will change when extensive changes are made.

Manual updates may be issued between editions to correct errors or document product changes. To ensure that you receive the updated or new editions, you should subscribe to the appropriate product support service. See your HP sales representative for details.

Edition 1: October 1992

Edition 2: January 1995

Preface

STREAMS/UX for the HP 9000 is Hewlett Packard's implementation of the AT&T de facto standard environment for communications protocols.

STREAMS/UX consists of the STREAMS environment, Transport Layer Interface (TLI), and XTI. TLI is an industry de facto standard application program interface for implementing transport-level communications by means of STREAMS-based network protocol stacks. HP also provides a Data Link Provider Interface (DLPI) adapter with the core operating system. DLPI is one industry standard definition for message communications to STREAMS-based network interface drivers.

This manual includes information on how to install STREAMS/UX, how to program with STREAMS/UX, and how to debug STREAMS/UX programs. The programming information in this manual is intended to be used in conjunction with the AT&T STREAMS manual called *UNIX System V Release 4 Programmer's Guide: STREAMS*.

This manual contains the following chapters:

Chapter 1 **Installation and Verification of STREAMS/UX** describes product installation using HP's installation and update program, *swinstall*, and describes how to verify the installation.

Chapter 2 **Detailed Product Information** provides a more in-depth explanation of the product installation, including instructions for manual kernel builds, information about STREAMS drivers and modules, and descriptions of STREAMS-related device files.

Chapter 3 **Differences Between STREAMS/UX and System V Release 4 STREAMS** summarizes differences in areas such as commands, system calls, utilities, drivers and data structures, and is intended primarily for programmers. Chapter 3 is written with the assumption that the programmer has already read the AT&T manual *UNIX System V Release 4 Programmer's Guide: STREAMS*.

Chapter 4 **STREAMS/UX Multiprocessor Support** discusses UP emulation, writing MP scalable modules and drivers, how to port SVR4 MP modules and drivers to HP-UX, and synchronization levels.

Chapter 5 **How to Compile and Link STREAMS/UX Drivers, Modules, and Applications** provides step-by-step instructions for each of these topics.

Chapter 6 **Debugging STREAMS/UX Modules and Drivers** provides a detailed look at how to use the *strdb* and *adb* debugging tools to debug STREAMS modules and drivers.

Chapter 7 **STREAMS/UX-NetTL Link** describes how STREAMS uses the Network Tracing and Logging facility.

1

Installation and Verification of STREAMS/UX

This chapter covers installation, configuration and verification of the STREAMS/UX subsystem for HP-UX systems, and consists of the following sections:

- System requirements
- STREAMS/UX filesets
- Loading STREAMS/UX software
- Configuring STREAMS-based pipes with the SAM program
- Verification of correct installation using *pdfck* and *strvf*

System Requirements

STREAMS/UX is installed and configured automatically during an HP-UX 10.0 installation.

STREAMS/UX does not require any dedicated hardware. Its drivers are all pseudo drivers.

STREAMS/UX is supported on all HP9000 Series 700 and 800 systems that HP-UX 10.0 supports.

STREAMS/UX Filesets

The HP-UX STREAMS product is organized into filesets. The filesets are organized by grouping together the files that make up the runtime environment, the kernel build components, and the manpages.

- STREAMS-RUN—Contains Transport Level Interface (TLI) library, X/Open Transport Interface (XTI) library, STREAMS/UX commands and STREAMS/UX user-space header files.
- STREAMS-MAN—Contains the STREAMS/UX man pages.
- STREAMS-KRN—Contains STREAMS/UX kernel library and kernel header files.

NOTE:

The library */usr/lib/libstr.a* provided as part of the HP-UX 9.0 STREAMS/UX product is no longer supplied as of HP-UX 10.0. The STREAMS/UX system calls for compiling STREAMS/UX applications are now part of the C libraries (for example, *libc.sl* and *libc.a*) as of HP-UX 10.0.

A fileset is a logical grouping of software files. HP uses this structure for organizing distribution of a product's software components. This fileset organization is then used by HP's installation program, *swinstall*, to load product files onto a system. For more information on *swinstall*, refer to the *Installing and Updating HP-UX* manual.

Loading STREAMS/UX Software

Follow the steps below to load STREAMS/UX software using the HP-UX *swinstall* program.

1. Insert the software media (tape or disk) into the appropriate drive.
2. Run the *swinstall* program using the command:

   `/usr/sbin/swinstall`
3. Enter the mount point of the drive in the Source Depot Path field, and activate the OK button to return to the Software Selection Window.

   The Software Selection Window now contains a list of available software to install.
4. Highlight the STREAMS/UX software. The “Selected” menu becomes active.
5. Choose Mark for Install from the “Selected” menu to choose the product to be installed.
6. Choose Install from the “Install” menu to begin product installation and open the Install Analysis Window.
7. Activate the OK button in the Install Analysis Window when the Status field displays a Ready message.
8. Activate the Yes button at the Confirmation Window to confirm that you want to install the software.

   swinstall loads the fileset, runs the customized scripts for the fileset, and builds the kernel.
9. Activate the OK button on the Note Window to return to the Install Window.
10. Activate the Show Logfile button to check for installation error messages. Refer to the message, cause and actions to correct any unresolved problems.
11. Activate the OK button in the Logfile Window to return to the Install Window.
12. Activate the OK button in the Install window to return to the Software Selection Window.
13. Choose Exit from the “File” menu to leave *swinstall*.

Configuring STREAMS-based Pipes with SAM

System Administration Manager (SAM) allows you to configure various tunable parameters. After installation is complete, all of the STREAMS/UX parameters are set to a default value and do not require any modifications. You may, however, want to change one tunable parameter. If you want to use STREAMS-based pipes, you will need to change this default value.

NOTE: By turning on STREAMS-based pipes, ALL of the pipes created by the pipe(2) command on the system will be STREAMS-based.

You can use SAM to configure STREAMS-based pipes. Follow the steps below:

1. In SAM, choose “Kernel Configuration,” followed by “Configurable Parameters.”
2. Highlight the “streampipes” label, then select “Modify Configurable Parameters” from the Actions menu.
3. Under the label “Choose One to Modify Parameters,” choose “Specify New Formula Value.” Set the formula value to 1 (one), then press OK.
4. In the File menu, choose exit. Before SAM exits, it will ask you when you want to have the new kernel created. Choose “Create a New Kernel Now.”
5. Press OK. The new kernel will be built and moved into place.

Verification of Correct Installation

Follow these steps to verify that the installation is correct:

1 Run the */usr/bin/pdfck* command to verify that the STREAMS/UX software was correctly installed on your system. Verification is done by checking a master product description file (*pdf*), which is delivered with the fileset, against the files just installed on the system. Run *pdfck* on each of the filesets that *swinstall* installed:

```
/usr/bin/pdfck  /system/STREAMS-KRN/pdf
/usr/bin/pdfck  /system/STREAMS-MAN/pdf (if fileset is installed)
/usr/bin/pdfck  /system/STREAMS-RUN/pdf
/usr/bin/pdfck  /system/STREAMS-PRG/pdf (if fileset is installed)
```

If the installation is correct, you should only receive a prompt after running the *pdfck* command. If *pdfck* finds a problem, it will report errors in the form of:

```
pathname: diff_field[(details)  ][  ,...]
```

where *diff_field* is one of the field names specified in pdf(4). The fields are *pathname*, *owner*, *group*, *mode*, *size*, *links*, *version*, *checksum*, and *linked_to*. Each field is separated by a colon (:). For more information, refer to the pdf(4), pdfdiff(1M) and pdfck(1M) man pages.

Any differences found by *pdfck* usually indicate installation problems. Verify that the STREAMS software was installed properly by reviewing steps 1 through 13 in the “Loading STREAMS/UX Software” section, and redo these steps if necessary.

2 To verify that STREAMS/UX software was properly configured into your HP-UX kernel, run the STREAMS verification tool, *strvf*, by typing:

```
/usr/sbin/strvf
```

If the STREAMS software has been properly installed and configured into the kernel, you should see the following messages:

```
-> Logging results to /var/adm/streams/strvf.log
-------------------------------------------------------
-> Verify HP Streams installation. Verify open, putmsg,  <-
-> getmsg, ioctl, and close can be performed on a stream.<-
-------------------------------------------------------
     ---------------------------------------------
     -> HP Streams is installed and operational <-
     ---------------------------------------------
```

If you wish, you can use the *verbose* (-v) option to receive information on what *strvf* is doing. *strvf* checks the following items:

- STREAMS kernel daemons are running.
- The echo driver (a core STREAMS driver) can be opened.
- a *putmsg*() can be performed on the *echo* driver.
- a *getmsg*() receives the same message sent by *putmsg()*.
- A STREAMS *ioctl* can be passed to the echo driver and acknowledged.
- The *echo* driver can be closed.

2

Detailed Product Information

This chapter provides a more in-depth explanation of the STREAMS/UX product installation than Chapter 1. The information provided here is primarily for reference.

This chapter contains information about core STREAMS drivers and modules, lists STREAMS-related tunables, and lists STREAMS-related device files.

Adding STREAMS Drivers and Modules

NOTE: The instructions below do not apply to clustered systems. If your system is attached to a cluster, follow the instructions in System Administration Tasks for Series 700 computers to configure the kernel. Alternatively, you can also create a new kernel using the SAM utility.

NOTE: Before attempting this procedure, familiarize yourself with the system reconfiguration information in the *mk_kernel(1M)* manual reference page and HP-UX system literature.

Refer to the System Administration manual for your system for complete instructions on how to create a kernel.

The software installation program, *swinstall*, usually builds a kernel correctly during product installation. In the unlikely event that the kernel is not built correctly, follow the steps below for manually building a STREAMS kernel.

The process involves modifying the kernel configuration input file to include the STREAMS subsystem, driver and module keywords.

Manual Kernel Build Procedures

If you used some other file to create the kernel previously, copy that file to */stand/system* before following the steps below.

1. Ensure that you have super-user capabilities.
2. Change to the */stand* directory.
3. Make a backup copy of your current configuration description file (which is commonly *system* or *build/system.SAM*).
4. Edit the *system* file to add drivers and/or change system parameters.

   hpstreams;

   dlpi;

   clone;

   strlog;

sad;

echo;

timod;

tirdwr;

ffs

pipemod

pipedev

sc;

5 Make a copy of the existing kernel (default name vmunix).

6 Regenerate the kernel with *mk_kernel*, using the edited *system* file as input. *mk_kernel* creates the new hp-ux kernel (the default is */stand/build/vmunnix_test*). There are two examples below. The first creates a new kernel in the build directory called *vmunix_test*. The second example automatically moves the kernel to the */stand* directory and makes a backup if the file, */stand/vmunix*, already exists.

```
mk_kernel
mk_kernel -s /stand/system -o /stand/vmunix
```

7 If you did not use the *-o* option with the *mk_kernel* command, copy the new kernel to */stand/vmunix*.

8 Reboot the new kernel. If the new kernel fails to boot, boot the system from the backup kernel and repeat the process of creating a new kernel. To do so, follow the instructions in your System Administration manual.

STREAMS Drivers and Modules

The configuration of STREAMS drivers and modules is statically defined at system creation time. The STREAMS subsystem, core drivers and modules are part of every 10.0 system.

The following sections contain a list of the core drivers and modules, STREAMS kernel tunable parameters, and STREAMS configuration data structure (streams_devs[ ]) information. See the master(4) manpage for more details.

STREAMS Drivers

The core STREAMS drivers are:

- clone—provides the device cloning used by STREAMS.
- strlog—provides the STREAMS logging facility.
- sad—provides the STREAMS module autopush capability.
- echo—loopback test driver used by the verification program, *strvf*. Refer to the strvf(1M) manpage.
- pipedev—required for STREAMS-based pipes.

STREAMS Modules

The core STREAMS modules are:

- sc—used by autopush and provides part of the STREAMS module autopush capability. Refer to the autopush(1M) manpage.
- timod—provides an interface from TLI/XTI to the transport provider.
- tirdwr—another TLI module; provides a read/write interface to the transport provider.
- pipemod—handles M_FLUSH messages for STREAMS-based pipes.

Miscellaneous STREAMS Functionality

- ffs—file system type required for fattach(3C).

Kernel Tunable Parameters

The following table describes STREAMS configurable parameters that are in */usr/conf/master.d/streams* file. The master file should not be modified. The values can be tuned using SAM.

Tunable Name	Default Value	Use
NSTREVENT	50	Determines the maximum number of outstanding STREAMS bufcalls allowed at any one instance. This needs to be modified if the protocol modules to be incorporated into STREAMS need to have more than 50 bufcalls outstanding at the same time.
STRMSGSZ	8192	Defines the maximum number of bytes that can be sent in the data part of a STREAMS message using the function *putmsg* and *write*. *Putmsg* will return ERANGE if a data buffer is sent with a size greater than this value. *Write* will segment the data into multiple messages. If STRMSGSZ is 0, the maximum data message size is infinite.
STRCTLSZ	1024	Defines the maximum number of bytes that can be sent in the control part of a STREAMS message using the function *putmsg*. *Putmsg* will return ERANGE if a buffer is sent with a size greater than this value. If STRCTLSZ is 0, the maximum control message size is infinite.
NSTRPUSH	16	Defines the maximum number of STREAMS modules that can be pushed onto a single stream.

Tunable Name	Default Value	Use

NSTRSCHED	0	Determines the number of streams scheduler daemons (smpsched) running on a MP system. The default value is 0, which indicates that Streams will determine the number of daemons based on the number of processors in the system. The number of MP streams schedulers created is as follows:
		
		`# of processors  # of smpscheds created`
		`--------------  --------------------`
		`2-4  1`
		`5-8  2`
		`8-16  3`
		`16+  4`
		
		If a tunable value > 0 is specified, then that value is used to determine the number of MP schedulers (smpsched) created on a MP system. The minimum value for this tunable is 0 and the maximum is 32.
		
		No MP schedulers will be created on a UP system.
		
		Also, regardless of whether a system is MP or UP, there will always be one UP Streams scheduler (supsched).
		
		NOTE: This tunable is for use by specific HP products only. It will likely be removed in future HP-UX releases.

NSTRBLKSCHED	2	Determines the number of blockable Streams scheduler daemons (sblksched) running on a MP system. The default value is 2, which means that two blockable Streams schedulers (sblksched) will be created on a MP system. If the tunable is set to 0, then no blockable Streams schedulers will be created on a MP system. Also, on a UP system, no blockable Streams schedulers will be created.
streamspipes	0	Determines if pipes are STREAMS-based. If set to zero, pipes are not STREAMS-based. If non-zero, pipes are STREAMS-based. The default is for pipes to not be STREAMS-based. NOTE: This tunable appears in */usr/conf/master.d/core-hpux*.

STREAMS-Related Device Files (Framework-specific)

This section lists the *mknod* commands necessary for manually creating device files. On a properly installed STREAMS system, these commands are not necessary. This section is included for informational purposes. All device files listed here are set-up to be STREAMS cloneable.

```
mknod /dev/strlog  c   72   0x49   #73   decimal
mknod /dev/sad     c   72   0x4a   #74   decimal
mknod /dev/echo    c   72   0x74   #116  decimal
```

3

Differences Between STREAMS/UX and System V Release 4 STREAMS

This chapter summarizes the differences between STREAMS/UX and System V Release 4.2 STREAMS. Chapter 4 discusses STREAMS/UX multiprocessor support and the differences between STREAMS/UX and System V Release 4 Multiprocessor STREAMS. You need to use this manual in conjunction with USL's *UNIX System V Release 4.2 STREAMS Modules and Drivers* and *UNIX System V Release 4.2 Device Driver Reference*. The USL manuals will be referred to as the SVR4.2 STREAMS manual and the SVR4.2 Driver manual from now on. Unless otherwise stated in this chapter and Chapter 4, STREAMS/UX information described in the SVR4.2 STREAMS and SVR4.2 Driver manuals will be applicable to STREAMS/UX.

NOTE:

This chapter is intended primarily for programmers, and is written with the assumption that you have already read the SVR4.2 STREAMS Modules and Drivers manuals.

Overview

This chapter will be divided into the following categories for describing differences between HP-UX and SVR4.2 STREAMS:

- Commands
- System calls
- Utilities
- Drivers and modules
- Data structures
- STREAMS/UX uniprocessor synchronization
- Cloning
- Hardware driver writing

HP-UX Changes to STREAMS/UX Commands

STREAMS/UX supports the commands listed below:

- autopush
- fdetach
- strace
- strchg
- strclean
- strconf
- strerr
- strvf

HP versions of supported STREAMS/UX commands operate somewhat differently from the way the commands are described in the *UNIX SVR4.2 Command Reference* manual. NLS catalogs exist for the commands. The catalogs are called *autopush.cat, fdetach.cat, strace.cat, strchg.cat, strclean.cat, strconf.cat, strerr.cat,* and *strvf.cat* and are located in the */usr/lib/nls/C* directory. Differences in the commands are described below.

autopush

The syntax for the autopush command on HP-UX is as follows:

```
autopush   -f   autopush_file_name
autopush   -r   -M   major_num|dev_name   -m   minor_num
autopush   -g   -M   major_num|dev_name   -m   minor_num

autopush_file_name contents:
major_num|dev_name low_minor high_minor mod_name 1...mod_name N
```

The HP-UX *autopush* command has been enhanced to allow the user to specify the device name in place of the major number, which is recommended since HP-UX provides dynamic major numbers. The name can be specified in the autopush file and on the command line. Device names are located in the HP-UX master files. The major number can still be used if needed.

strace and strerr

The *strace* and *strerr* commands use the STREAMS log driver, */dev/strlog*. SVR4.2 calls this driver */dev/log*, but HP-UX already includes a non-streams driver named */dev/log*.

HP-UX Changes to STREAMS/UX System Calls

NOTE:

By default HP-UX terminal I/O is not implemented using STREAMS/UX in HP-UX 10.0. But a STREAMS-based *pty* is available in the STREAMS-TIO offering included in the HP-UX runtime product.

STREAMS/UX supports the following system calls:

- close
- fattach
- fcntl
- fdetach
- getmsg
- getpmsg
- ioctl
- isastream
- open
- pipe
- poll
- putmsg
- putpmsg
- read
- readv
- select
- signal
- write
- writev

For STREAMS-based termio, see the following manpages (which are part of the STREAMS-TIO product): grantpt(3C), ptsname(3C), and unlockpt(3C).

There are HP-UX modifications to the *fattach*, *ioctl*, *pipe*, *poll*, *putmsg*, *putpmsg*, *select*, *signal*, *write*, and *writev* system calls. These modifications are as follows.

fattach Modifications

STREAMS/UX supports the *fattach(3)* and *fdetach(3)* library calls and the *fdetach(1m)* command as described in the *UNIX SVR4.2 Operating System API Reference* and the SVR4.2 Command Reference. In order to use *fattach* and *fdetach*, the kernel must have the *ffs* file system configured. *ffs* is added to the */stand/system* file when STREAMS/UX is installed using *swinstall*. If *ffs* has been deleted after the install was done, re-include it as follows, regenerate a kernel, and reboot the system.

```
ffs
```

ioctl Modifications

STREAMS/UX supports *ioctl* as described in the SVR4.2 STREAMS manual.

Also, note that the multiplexor ID number returned by I_LINK and I_PLINK is a memory address, not a small integer such as 0, 1, 2, 3.

pipe Modifications

STREAMS/UX supports STREAMS-based pipes as an optional feature. STREAMS/UX's STREAMS-based pipes behave as described in the *UNIX SVR4.2 Operating System API Reference* and the *UNIX System V Release 4 Programmer's Guide: STREAMS*.

By default, pipes created by the *pipe(2)* system call are not STREAMS-based. In order to get STREAMS-based pipes, the */stand/system* file must have the pipemod and pipedev module and driver configured, and the tunable parameter streampipes must be set to 1 (one).

When STREAMS/UX is installed, the */stand/system* file is modified to include *pipemod* and *pipedev*, but streampipes is set to zero by default. The kernel must be regenerated and the system rebooted if the setting of streampipes to non-zero is to take effect. In other words, *adb'ing* the running system to turn streampipes on will have no effect on the type of pipes created by *pipe(2)*. Once the kernel is regenerated and rebooted, all *pipe(2)*

pipes on the system will be STREAMS-based. However, fifos will not be STREAMS-based. STREAMS/UX does not support STREAMS-based fifos.

The STREAMS/UX device pipedev is only for internal STREAMS/UX use in implementing STREAMS-based pipes. Opening a device file with *pipedev*'s major number will not result in a STREAMS-based pipe, or even a properly functioning stream. STREAMS-based pipes must be created using the *pipe(2)* system call.

PIPE_BUF is a pathname variable value, and SVID, XPG4, POSIX, etc. define it as the maximum number of bytes that is guaranteed to be written atomically. To obtain the correct value of PIPE_BUF, use fpathconf() (see pathconf()). For STREAMS-based pipes, the value of PIPE_BUF depends on the configurable parameter STRMSGZ (by default, 8KB). For example, PIPE_BUF is set to 4KB if STRMSGSZ is 4KB, 8KB if STRMSGSZ is 8KB, and 16KB if STRMSGSZ is 16KB. There is one exception. If STRMSGSZ is set to 0 (i.e. infinite size), then PIPE_BUF for STREAMS/UX pipes is set to 8KB.

putmsg and putpmsg Modifications

Maximum and Minimum Data Buffer Size

The size of the user's data buffer must be within the minimum and maximum packet size range specified in the topmost STREAM module's streamtab. It must also be less than or equal to STRMSGSZ. If the number of bytes to transfer is not in this range, ERANGE will be returned.

Maximum and Minimum Control Buffer Size

The size of the user's control buffer must be less than or equal to both STRCTLSZ and STRMSGSZ. If STRCTLSZ is less than or equal to zero, the page size is used instead of STRCTLSZ for this check.

Data Buffer Segmentation

The user's data buffer may be sent in multiple data blocks chained together to form a message. The maximum number of bytes, including the write offset, that can be sent in one data block is equal to the page size.

Write Offset

A module or driver can send the stream head an M_SETOPTS message, telling the STREAM head to put an offset in the beginning of the first data block in a message sent by a *putmsg* call. STREAMS/UX will not put the offset into the data block if the amount of memory required is greater than the page size. See Chapter 5 of the SVR4.2 STREAMS manual for more information.

select Modifications

STREAMS/UX supports the *select* system call for STREAMS/UX devices. For information about the *select* system call, see the select(2) man page delivered with the HP-UX core system.

The *select* system call does not provide as much information as *poll*. If *select* returns an event for a STREAMS/UX device, the program can call *poll* to get more information.

A select read event is returned if a poll event POLLRDNORM, POLLERR, POLLNVAL or POLLHUP exists on the stream. In other words, a read event is returned for the following conditions:

- a normal message is waiting to be read
- a read error exists at the stream head
- a write error exists at the stream head
- the stream is linked under a multiplexor
- a hang-up has occurred

A select write event is returned if a poll event POLLOUT, POLLWRNORM, POLLERR, POLLNVAL, or POLLHUP exists on the STREAM. This means that a write event is returned for the following conditions:

- normal data can be written without blocking because of flow control
- a read error exists at the stream head
- a write error exists at the stream head
- the stream is linked under a multiplexor
- a hang-up has occurred

A select exception event is returned if a poll event POLLPRI or POLLRDBAND exists on the STREAM. More specifically, an exception event is returned if a high-priority message or a banded message is waiting to be read.

signal Modifications

STREAMS/UX supports signals and the HP-UX *signal* system call. However, STREAMS/UX does not support extended signals or the *siginfo_t* structure described in the siginfo(5) manpage.

write and writev Modifications

Maximum and Minimum Data Buffer Size

The size of the user's data buffer must be within the minimum and maximum packet size range specified in the topmost STREAM module's streamtab. If the number of bytes to transfer is not in this range, ERANGE will be returned. Two exceptions exist in which no error occurs. The first exception is if the data buffer is too large and either the maximum packet size is infinite or the minimum packet size is less than or equal to zero. (An infinite packet size is specified using the define INFPSZ in the *stream.h* file.) The second exception occurs if the buffer is too small and the minimum packet size is less than or equal to zero. With either exception, ERANGE is not returned, and the data is transferred.

Data Buffer Segmentation

The user's data buffer may be sent in multiple messages. The maximum amount of data that can be sent in one message is the lower value of the topmost module's maximum packet size and STRMSGSZ. If the maximum packet size is infinite, then the top module’s high water mark is taken into consideration. If the high water mark is more than zero, half of the high water mark is used; otherwise the page size is used.

Write Offset

A module or driver can send the STREAM head an M_SETOPTS message telling it to put an offset in the beginning of each data buffer segment (i.e. message) sent by a write call. See Chapter 5 of the SVR4.2 STREAMS manual for more information. STREAMS/UX will not put the offset into a message if the resulting message size exceeds STRMSGZ.

HP-UX Modifications to STREAMS/UX Utilities

STREAMS/UX supports the following kernel utilities described in the SVR4.2 Driver manual, although some of the utilities have been modified for HP-UX.

adjmsg
allocb
backq
bcanput
bcanputnext
bcopy
bufcall
bzero
canput
canputnext
cmn_err
copyb
copymsg
datamsg
delay
drv_getparm
drv_priv
dupb
dupmsg
enableok
esballoc
esbbcall
flushband
flushq
freeb
freemsg
freezestr
getadmin
getmid
getmajor
getminor
getq
insq
itimeout
kmem_alloc
kmem_free
linkb
LOCK
LOCK_ALLOC
LOCK_DEALLOC
major
makedev
makedevice
max
min
minor
msgdsize
msgppullup
noenable
OTHERQ
pcmsg
pullupmsg
put
putbq
putctl
putctl1
putnext
putnextctl
putnextctl1
putq
qenable
qprocsoff
qprocon
qreply
qsize
RD
rmvb
rmvq
SAMESTR
sleep
spln
splstr
strlog
strqget
strqset
SV_ALLOC
SV_BROADCAST
SV_DEALLOC
SV_WAIT
SV_WAIT_SIG
testb
timeout
TRYLOCK
unbufcall
unfreezestr
unlinkb
UNLOCK
untimeout
vtop
wakeup
WR

In addition, HP-UX provides the following new utilities.

get_sleep_lock
putctl2
putnextctl2
streams_put
unweldq
weldq

The *strenv.h* file redefines some native HP-UX kernel utilities to conform to System V Release 4.2. The *strenv.h* file redefines *delay, get_sleep_lock, kmem_alloc, kmem_free, lbolt, max, min, sleep, time, timeout,* and *untimeout*. These defines might collide with declarations in STREAMS/UX modules and drivers. You can customize the *strenv.h* file to avoid collisions or to use native HP-UX utilities. However, modules and drivers cannot call the native HP-UX *sleep* or *get_sleep_lock* directly. If your modules and drivers call *sleep* or *get_sleep_lock*, you must include *strenv.h* to redefine *sleep* and *get_sleep_lock* to *streams_mpsleep* and *streams_get_sleep_lock*. For more information about the native HP-UX primitives, see the *HP-UX Driver Development Guide*, part number 98577-90014.

Differences between the STREAMS/UX kernel utilities and the descriptions in the SVR4.2 Driver manual are discussed below, along with information about new utilities. This section assumes that modules and drivers include *strenv.h*.

esballoc

The STREAMS/UX *esballoc* is the same as the *esballoc* call described in the SVR4.2 Driver manual with a few differences. The HP-UX *esballoc* copies the contents of the *fr_rtn* structure into an area of the data block not visible to the STREAMS/UX programmer. Then *esballoc* stores a pointer to this area in the *db_freep* field. This allows modules and drivers to modify the *fr_rtn* parameter after calling *esballoc* without affecting subsequent *freeb* calls. Also, modules and drivers can change a data block's *fr_rtn* information by modifying the structure pointed to by *db_freep*. The free routine passed to *esballoc* can call STREAMS/UX utilities in the same way as the *put* or *service* routine that called *freeb*. Also, a free routine can safely access the same data structures as the put or service routine that called *freeb*. However, unlike SVR4.2, HP-UX does not block interrupts from all STREAMS/UX devices while the free routine runs. See “STREAMS/UX

Uniprocessor Synchronization" in this chapter and "Writing MP Scalable Modules and Drivers" in Chapter 4 for more information about *esballoc* free routines.

cmn_err

The STREAMS/UX *cmn_err* is the same as the cmn_err described in the SVR4.2 Driver manual with a few differences. The HP-UX *cmn_err* always sends messages to both the system console and the circular kernel buffer. Inserting an exclamation point ("!") or a circumflex ("^") as the first character in the format string has no effect. HP-UX simply removes these control characters from the message, and sends the message to both the console and the kernel buffer. There are a couple of other very minor differences. HP-UX precedes CE_PANIC level messages with the string **`panic:`** instead of **`PANIC:`**. Also, the HP-UX circular kernel buffer is called msgbuf instead of putbuf. The HP-UX *msgbuf* is a fixed size, and can be viewed using the *dmesg* command or the *adb* debugger tool.

freezestr and unfreezestr

The SVR4.2 Driver manual says that *freezestr* and *unfreezestr* must be called on multiprocessors to protect searching a STREAMS/UX queue and calling *insq*, *rmvq*, *strqset*, and *strqget*. SVR4 MP provides *freezestr* and *unfreezestr* to prevent software on multiple processors from manipulating a queue's list of messages at the same time. STREAMS/UX uses synchronization levels for this. See "Writing MP Scalable Modules and Drivers" in Chapter 4 for more information about synchronization levels and HP-UX limitations on *insq*, *rmvq*, *strqset*, and *strqget*. Because STREAMS/UX uses a different mechanism to protect STREAMS/UX queues, the HP-UX *freezestr* just returns the current interrupt priority level, and *unfreezestr* is a no-op. HP-UX provides the *freezestr* and *unfreezestr* stubs to make porting code from SVR4 MP easier.

get_sleep_lock

STREAMS/UX provides some extra support for modules and drivers which use the native HP-UX *get_sleep_lock* primitive. Alternatively, modules and drivers can call the SVR4 MP SV_WAIT and SV_WAIT_SIG. Open and close routines call *get_sleep_lock* before sleeping to prevent missing wakeups. After calling *get_sleep_lock*, the *open* or *close* can release

spinlocks before sleeping. Other processes cannot wakeup the *open* or *close* between the time it calls *get_sleep_lock* and *sleep*. Modules and drivers must include *strenv.h* to use *get_sleep_lock. strenv.h* redefines *get_sleep_lock* to *streams_get_sleep_lock*. Modules and drivers cannot call the native HP-UX *get_sleep_lock* directly, because STREAMS/UX needs to do some additional synchronization before invoking *get_sleep_lock*.

```
lock_t *
get_sleep_lock(event);

       caddr_t event;
```

The *open* or *close* routine passes the event it will pass to the sleep primitive to *get_sleep_lock*. *get_sleep_lock* obtains a sleep spinlock, and returns a pointer to this lock.

itimeout

If the HP-UX *itmeout* cannot allocate memory, it panics instead of returning 0 like the SVR4 MP *itimeout*. The STREAMS/UX *itimeout* only returns 0 if it is passed an interrupt priority level that is lower than *pltimeout*. You can increase the amount of memory available to both the new *itimeout* and the existing timeout primitives using the NCALLOUT tunable. Set NCALLOUT to the maximum number of *itimeout* and *timeout* requests that can be outstanding at any one time.

kmem_alloc

The STREAMS/UX *kmem_alloc* tries to allocate 32 bytes if the size parameter is set to 0. The SVR4.2 *kmem_alloc* returns NULL instead.

LOCK

The STREAMS/UX LOCK calls the native HP-UX spinlock primitive. LOCK has an interrupt priority level parameter, which is used to raise the priority level and block interrupts which acquire the spinlock. The SVR4.2 Driver manual says that implementations which do not need to raise the interrupt level can ignore this parameter. Since the HP-UX spinlock primitive always raises the interrupt level to spl6 while a spinlock is held, STREAMS/UX ignores the interrupt level parameter on multiprocessor systems. For better performance on uniprocessor systems, the STREAMS/UX LOCK raises the priority level to the parameter value

instead of acquiring a spinlock. Whether the caller will block or spin if the lock cannot be obtained is implementation defined. The HP-UX implementation spins.

LOCK_ALLOC

The STREAMS/UX LOCK_ALLOC calls the native HP-UX *alloc_spinlock* primitive. There are some small differences between the STREAMS/UX LOCK_ALLOC and the SVR4 MP utility. LOCK_ALLOC has a flag parameter which indicates if the caller is willing to block while waiting for memory to be allocated. HP-UX only allows this flag to be set to KM_SLEEP, and returns zero if it is set to KM_NOSLEEP. The STREAMS/UX LOCK_ALLOC accepts the following hierarchy parameter values which are reserved for STREAMS/UX modules and drivers in */usr/include/sys/semglobal.h* and */usr/conf/h/semglobal.h:* STREAMS_USR1_LOCK_ORDER, STREAMS_USR2_LOCK_ORDER, and STREAMS_USR3_LOCK_ORDER. The compiler options to turn on deadlock checking for HP-UX are different than those documented in the SVR4.2 Driver manual. The entire HP-UX kernel and the module or driver must be compiled with SEMAPHORE_DEBUG to enable deadlock checking. According to the SVR4.2 Driver manual, the min_pl parameter can be ignored by implementations which do not need to raise the priority level. The HP-UX STREAMS LOCK_ALLOC ignores it.

putctl2

STREAMS/UX also provides the additional utility called *putctl2*. This utility can be used to send a control message with a two-byte parameter to a queue. For example, *putctl2* can send the new style of an M_ERROR message, which is two bytes long, to a queue.

```
int putctl2(q, type, p1, p2);

      queue_t *  q;
      int        type;
      int        p1;
      int        p2;
```

The q parameter is the queue to which the message is sent. The type parameter is the message type. The *p1* and *p2* parameters are the two bytes of data in the message. The *putctl2* utility ensures that the type is not a data type. The utility also allocates a message block, fills in the data, and calls

the put routine of the specified queue. *putctl2* returns 0 if the type is M_DATA, M_PROTO or M_PCPROTO, or if a message block cannot be allocated. *putctl2* returns 1 if it completes successfully.

putnextctl2

STREAMS/UX provides the additional utility *putnextctl2*. This utility can be used to send a control message with a two-byte parameter to the next queue in a stream. For example, *putnextctl2* can send the new style of an M_ERROR message, which is two bytes long, to the next queue in a stream.

```
int putnextctl2(q, type, p1, p2);

    queue_t * q;
    int       type;
    int       p1;
    int       p2;
```

The *q* parameter is the queue from which the message is sent. The message is sent to *q->q_next*. The type parameter is the message type. The *p1* and *p2* parameters are the two bytes of data in the message. The *putnextctl2* utility ensures that the type is not a data type. The utility also allocates a message block, fills in the data, and calls the put routine of *q->q_next*. *putnextctl2* returns 0 if the type is M_DATA, M_PROTO, or M_PCPROTO, or if a message block cannot be allocated. *putnextctl2* returns 1 if it completes successfully.

qprocson and qprocsoff

SVR4 MP STREAMS/UX provides *qprocson* and *qprocsoff*, which on a multiprocessor system allows a module's *put* and *service* routines to run concurrently with *open* and *close*. STREAMS/UX does not allow this much parallelism. A module's or driver's *put* and *service* routines cannot run at the same time as the *open* or *close*. Although STREAMS/UX does not run the *put* or *service* routine in parallel with the *open* or *close*, it does queue any requests to run the *put* or *service* routine. STREAMS/UX will process these when *open* finishes. Also, if *open* or *close* sleeps, STREAMS/UX can run the *put* and *service* routines while *open* or *close* are sleeping. However, a *put* or *service* routine cannot do the wakeup on a sleeping *open* or *close*. STREAMS/UX provides stubs which are no-ops for *qprocson* and *qprocsoff* to make porting easier.

streams_put utilities

STREAMS/UX provides a new utility *streams_put*, which allows non-STREAMS/UX software to safely call STREAMS/UX utilities. timeout and bufcall user functions and other non-STREAMS/UX code cannot call several of the STREAMS/UX utilities or share data with modules and drivers. For a more detailed discussion about these restrictions, see "STREAMS/UX Uniprocessor Synchronization" in this chapter and "Writing MP Scalable Modules and Drivers" in Chapter 4.

Non-STREAMS/UX code can call *streams_put*, passing it a function and a queue. STREAMS/UX runs the function as if it were the queue's *put* routine. The function can safely manipulate the queue and access the same data structures as the queue's put routine.

```
#ifdef _PROTOTYPES
typedef void (*streams_put_t)(void *, MBPKP);
#else
typedef void (*streams_put_t)();
#endif

void
streams_put(func, q, mp, private)
            streams_put_t  func;
            queue_t        *q;
            mblk_t         *mp;
            void           *private;
```

STREAMS/UX will run *func* as if it were *q*'s *put* routine. STREAMS/UX passes *private* and *mp* to *func*. The non-STREAMS/UX code can pass any value in the private parameter. The code must pass a valid message block pointer in mp. *streams_put* uses fields in the message block not visible to the STREAMS/UX programmer.

SV_WAIT

STREAMS/UX implements a subset of the SVR4 MP synchronization variable utilities using sleep and wakeup. The HP-UX SV_WAIT differs from the SVR4 MP utility in the following ways. When the SVR4 MP SV_WAIT returns, the lkp spinlock is not held, and the priority level is set to plbase (SPLNOPREEMPT on HP-UX). On a multiprocessor system, the HP-UX SV_WAIT lowers the priority level to the value before the caller acquired the *lkp spinlock*, which may not be SPLNOPREEMPT. If the caller acquired the lock while holding other spinlocks, the priority level is lowered to the value before the first of these nested spinlock calls. Also, the SVR4

MP SV_WAIT has a priority argument that specifies the priority the caller would like to run at after waking. Since the HP-UX SV_WAIT is implemented by calling *sleep*, the HP-UX priorities are different than the SVR4 MP ones. On HP-UX, the priority passed into SV_WAIT is subtracted from PZERO-1. *pridisk, prinet, pritty, pritape, prihi, primed,* and *prilo* are defined to be 0, and do not affect the caller's priority. If you need to change the process's priority, study the priorities in */usr/include/sys/param.h* or */usr/conf/h/param.h*, and pass the needed offset to PZERO-1 in the priority parameter.

SV_WAIT_SIG

STREAMS/UX implements a subset of the SVR4 MP synchronization variable utilities using *sleep* and *wakeup*. The HP-UX SV_WAIT_SIG differs from the SVR4 MP utility in the following ways. When the SVR4 MP SV_WAIT_SIG returns, the *lkp spinlock* is not held, and the priority level is set to *plbase* (SPLNOPREEMPT on HP-UX). On a multiprocessor system, the HP-UX SV_WAIT_SIG lowers the priority level to the value before the caller acquired the *lkp spinlock*, which may not be SPLNOPREEMPT. If the caller acquired the lock while holding other spinlocks, the priority level is lowered to the value before the first of these nested spinlock calls. Also, the SVR4 MP SV_WAIT_SIG has a priority argument that specifies the priority the caller would like to run at after waking. Since the HP-UX SV_WAIT_SIG is implemented by calling *sleep*, the HP-UX priorities are different than the SVR4 MP ones. On HP-UX, the priority passed into SV_WAIT_SIG is added to PZERO+1|PCATCH. *pridisk, prinet, pritty, pritape, prihi, primed,* and *prilo* are defined to be 0, and do not affect the caller's priority. If you need to change the process's priority, study the priorities in */usr/include/sys/param.h* or */usr/conf/h/param.h*, and pass the needed offset to PZERO+1|PCATCH in the priority parameter. The last difference is that the SVR4 MP SV_WAIT_SIG returns if the process is first stopped by a job control signal and then continued. The HP-UX SV_WAIT_SIG continues to sleep until it receives a signal which does not stop the process, or an SV_BROADCAST wakes up the process.

TRYLOCK

The STREAMS/UX TRYLOCK calls the native HP-UX *cspinlock* primitive. TRYLOCK has an interrupt priority level parameter, which is used to raise the priority level and block interrupts which acquire the spinlock. The SVR4.2 Driver manual says that implementations which do not require the interrupt level to be raised can ignore this parameter. STREAMS/UX ignores the parameter on multiprocessor systems since the HP-UX *cspinlock* primitive always raises the interrupt level to *spl6* while a spinlock is held. For better performance on uniprocessor systems, the STREAMS/UX TRYLOCK raises the priority level to the parameter value instead of acquiring a spinlock.

UNLOCK

The STREAMS/UX UNLOCK calls the native HP-UX *spinunlock* primitive. UNLOCK has an interrupt priority level parameter, which is used to lower the priority level. HP-UX will ignore this parameter on multiprocessor systems. If the caller is not holding any other spinlocks, the STREAMS/UX UNLOCK lowers the priority level to the value before the caller acquired the spinlock. On uniprocessor systems, the STREAMS/UX UNLOCK lowers the priority level to the parameter value instead of releasing a spinlock.

weldq and unweldq

STREAMS/UX provides the additional utilities weldq and unweldq to allow the user to build a pipe-like stream. These utilities are provided because the programmer is not allowed to modify *q_next* pointers directly. This restriction and others are described in more detail in the section called “HP-UX Changes to STREAMS/UX Data Structures.”

unweldq

The utility *unweldq* disconnects two drivers' queues that were joined by *weldq*:

```
int unweldq (d1_wq, d2_rq, d2_wq, d1_rq, func, arg, protect_q);

       queue_t *     d1_wq;
       queue_t *     d2_rq;
       queue_t *     d2_wq;
       queue_t *     d1_rq;
       weld_fcn_t    func;
       weld_arg_t    arg;
       queue_t *     protect_q;
```

d1_wq and *d1_rq* are one of the driver's write and read queues. *d2_wq* and *d2_rq* are the second driver's queues. *unweldq* will set *d1_wq->q_next* and *d2_wq->q_next* to zero. Also, it updates queue fields used for flow control that are not visible to the STREAMS/UX programmer, and therefore cannot be changed by the STREAMS/UX programmer.

unweldq returns to the caller before disconnecting the drivers. *unweldq* requests that the STREAMS/UX *weld* daemon update the queues.

Note that if one end of a pipe-like stream created by *weld* is closed, STREAMS/UX will automatically unweld the two drivers. *unweldq* does not need to be called.

The weld daemon will call *func* with *arg* as an argument after it finishes the request. *protect_q* specifies which queue the callback function can access safely. See “STREAMS/UX Uniprocessor Synchronization” in this chapter and “Writing MP Scalable Modules and Drivers” in Chapter 4 for a more detailed discussion of *protect_q*.

If your driver does not need to be notified when the daemon finishes the *weld* request, pass *weldq* zero for the *func*, *arg*, and *protect_q* parameters.

On successful completion, *unweldq* returns 0. Otherwise, it returns an *errno* indicating the type of error that occurred. One of the following three values will be returned:

- ENXIO indicates that the weld daemon is not running.
- EINVAL indicates that invalid queue arguments are present.
- EAGAIN means that no memory is available.

weldq

Weldq connects two drivers' queues to form a pipe by setting the *q_next* pointer:

```
int weldq (d1_wq, d2_rq, d2_wq, d1_rq, func, arg, protect_q);

       queue_t *     d1_wq;
       queue_t *     d2_rq;
       queue_t *     d2_wq;
       queue_t *     d1_rq;
       weld_fcn_t    func;
       weld_arg_t    arg;
       queue_t *     protect_q;
```

d1_wq and *d1_rq* are one of the drivers' write and read queues. *d2_wq* and *d2_rq* are the second driver's queues. *weldq* will set *d1_wq->q_next* to be *d2_rq* and *d2_wq->q_nex*t to *d1_rq*. Also, *weldq* updates queue fields used for flow control that are not visible to the STREAMS/UX programmer, and therefore cannot be updated by the STREAMS/UX programmer.

weldq returns to the caller before connecting the drivers. *weldq* requests the STREAMS/UX weld daemon to update the queues.

The *weld* daemon will call *func* with *arg* as an argument after it finishes the request. *protect_q* specifies which queue the callback function can access safely. See “STREAMS/UX Uniprocessor Synchronization” in this chapter and “Writing MP Scalable Modules and Drivers” in Chapter 4 for a more detailed discussion of *protect_q*.

If your driver does not need to be notified when the daemon finishes the weld request, pass *weldq* zero for the *func*, *arg*, and *protect_q* parameters.

On successful completion, *weldq* returns 0. However, if *weldq* fails, an *errno* indicating the type of error that has occurred is returned. The *errno* will contain one of the following three values:

- ENXIO means that the weld daemon is not running.
- EINVAL means that invalid queue arguments exist.
- EAGAIN means that no memory is available.

Note that if one end of a pipe-like stream created by *weldq* is closed, STREAMS/UX will automatically unweld the two drivers. *unweldq* does not need to be called.

vtop

The STREAMS/UX *vtop* only accepts a NULL process structure pointer. In other words, it only converts kernel space addresses.

HP-UX Changes to STREAMS/UX Drivers and Modules

The unsupported drivers and modules include:

- connld
- console
- ports
- sxt
- xt

NOTE: Some STREAMS-based terminal I/O functionality is contained in a separate product called STREAMS-TIO. It is part of the HP-UX runtime product. See the following manpages (which are part of the STREAMS-TIO product): pts(7), ptm(7), ldterm(7), pterm(7), and pckt(7).

STREAMS/UX provides the following drivers and modules:

- clone
- strlog
- sad
- echo
- sc
- timod
- tirdwr
- pipemod

Entries for these drivers and modules can be found in the STREAMS/UX master file. General information about these drivers follows. Information about the stream head is also included. Differences between the HP-UX and SVR4.2 *log* and *sad* drivers are also described.

NOTE: Any driver or module not explicitly listed as supported in this section is not supported.

clone

Major Number: 72

clone is used to provide cloning. The major number of the device file for a cloneable driver must be the clone driver's major number, 72. The minor number is set to the real major number of the device.

strlog

Major Number: 73

Module ID Number: 44

Maximum Packet Size: INFPSZ

Minimum Packet Size: 0

High Water Mark: 2048

Low Water Mark: 128

The STREAMS/UX log driver is named *strlog* instead of *log*. The special device file is */dev/strlog*. *strlog* provides the same functionality for logging as described in the *UNIX SVR4.2 System Files and Devices Reference*, with the exceptions described below:

- The strlog kernel utility formats binary arguments before sending messages up the stream.
- STREAMS/UX does not provide a separate console logger or */dev/console* device. strlog does not support the I_CONSLOG ioctl. *strlog* prints a log message on the console if the SL_CONSOLE flag is set.
- The HP-UX *log_ctl* structure does not contain a *pri* field. Priority and facility codes are not supported.

sad

Major Number: 74

Module ID Number: 45

Maximum Packet Size: INFPSZ

Minimum Packet Size: 0

High Water Mark:	2048
Low Water Mark:	128

The HP-UX sad driver device file is */dev/sad*. The system administrator and users can open */dev/sad*. However, only the system administrator can execute the SAD_SAP *ioctl* system call. This differs from the System V sad driver, which is accessed through the */dev/sad/admin* and */dev/sad/user* device files.

sad provides autopush functionality as described in the *UNIX SVR4.2 System Files and Devices Reference* manual.

echo

Major Number:	116
Module ID Number:	5000
Maximum Packet Size:	INFPSZ
Minimum Packet Size:	0
High Water Mark:	2048
Low Water Mark:	128

echo is a loopback driver used by the *strvf* STREAMS/UX verification tool. For more information about *strvf*, see Chapter 1.

sc

Module ID Number:	5002
Maximum Packet Size:	INFPSZ
Minimum Packet Size:	0
High Water Mark:	2048
Low Water Mark:	128

sc provides auxiliary functions for the *sad* driver.

timod

Module ID Number:	5006
Maximum Packet Size:	INFPSZ
Minimum Packet Size:	0
High Water Mark:	2048
Low Water Mark:	128

timod provides TLI functionality as described in the *UNIX SVR4.2 System Files and Devices Reference* manual.

tirdwr

Module ID Number:	0
Maximum Packet Size:	INFPSZ
Minimum Packet Size:	0
High Water Mark:	16K
Low Water Mark:	128

tirdwr provides an alternative interface to the TLI library for accessing a transport protocol provider. *tirdwr* is described in the *UNIX SVR4.2 System Files and Devices Reference* manual.

Stream Head

Module ID Number:	0
Module Name:	sth
Maximum Packet Size:	INFPSZ
Minimum Packet Size:	0
High Water Mark:	10240
Low Water Mark:	1024

The Stream head provides the interface between HP-UX system calls and STREAMS/UX utilities in the kernel. The Stream head is the first queue pair of every Stream and is involved in flow control. Data being read from a stream will be taken off the stream head.

pipemod

Module ID Number:	5303
Maximum Packet Size:	8192
Minimum Packet Size:	0
High Water Mark:	8192
Low Water Mark:	8191

pipemod handles M_FLUSH messages in STREAMS/UX-based pipes. *pipemod* is described in the *UNIX System V Release 4 Programmer's Guide: STREAMS* manual.

HP-UX Changes to STREAMS/UX Data Structures

STREAMS/UX data structures are almost identical to those described in the SVR4.2 Driver manual. STREAMS/UX places additional restrictions on how some of these structures can be accessed.

STREAMS/UX data structures that differ from the descriptions in the SVR4.2 Driver manual are described below. Data structures identical to those described in the SVR4.2 manual are not listed below.

STREAMS/UX data structures contain some declarations for fields used by STREAMS/UX internally that are not visible to the STREAMS/UX programmer. The programmer will not be affected by these fields except that the *sizeof* function will return a larger value.

Message Structures

These structures are slightly different from the ones in the SVR4.2 Driver manual.

msgb

This structure is defined in the file *stream.h*.

The *msgb* structure contains MSG_KERNEL_FIELDS, which defines fields used internally by STREAMS/UX.

iocblk

The *iocblk* structure is defined in *stream.h*.

ioc_count is defined to be a member of a union.

copyreq

The *copyreq* structure is defined in *stream.h*.

cq_addr is defined to be a member of a union.

copyresp

The *copyresp* structure is defined in *stream.h*.

cp_rval is defined to be a member of a union.

Queue Structure

The queue structure is slightly different from the one described in the SVR4.2 Driver manual. The structure is defined in the file *stream.h*.

QUEUE_KERNEL_FIELDS defines fields used internally by STREAMS/UX.

STREAMS/UX Data Structure Restrictions

STREAMS/UX has the same restrictions as those described in the Kernel Data Structure chapter of the SVR4.2 Driver manual. Also, STREAMS/UX limits which user written functions can access the queue structure directly. A queue's *open*, *close*, *put*, or *service* routine can manipulate the queue structure as specified by SVR4.2. On a uniprocessor system, a queue's entry points can access the other queue in the queue pair in the same way that they can access their own queue. On a multiprocessor system, a queue's entry points can manipulate queues belonging to entities with which they can share data. They can manipulate the queues in the same way that they can manipulate their own queue. See “Writing MP Scalable Modules and Drivers” in Chapter 4 for more information about sharing data on multiprocessor systems.

It is difficult to program other functions (besides those described above) to access the queue structure directly, especially on multiprocessor systems. If a queue's entry points access queues other than those described above, or if non-STREAMS/UX software processes data in a STREAMS/UX queue, try to use the *streams_put* utility to manipulate the queues safely. *streams_put* is described in the “HP-UX Modifications to STREAMS/UX Utilities” section of this chapter. If you cannot use *streams_put*, the code that accesses a STREAMS/UX queue must, at a minimum, follow these additional rules. The software must ensure that it is accessing an allocated, opened queue. Also, it cannot dereference the *q_first*, *q_last*, or *q_nex*t pointers. In other words, it cannot read or write data pointed at by the pointers. For example, the function can check if *q_first* is 0, but it cannot read the *q_first-b_next* field. Lastly, you must implement any additional synchronization required for your modules and drivers to work correctly. You may need to synchronize the function accessing the STREAMS/UX queue with the queue's entry points. This is because the function and the entry points may access the queue in parallel on a multiprocessor system and may interrupt each other while accessing the queue on a uniprocessor system.

STREAMS/UX Uniprocessor Synchronization

This section describes STREAMS/UX synchronization on a uniprocessor system. Chapter 4 discusses multiprocessor synchronization. Also, Chapter 4 describes how modules and drivers running on a uniprocessor system can use multiprocessor synchronization mechanisms to protect against interrupts. STREAMS/UX programmers must follow the guidelines listed below as well as those in the SVR4.2 STREAMS manual.

STREAMS/UX provides the following types of synchronization on a uniprocessor system:

- STREAMS/UX protects its internal data structures from interrupts.
- STREAMS/UX helps protect module and driver private data structures against interrupts.
- STREAMS/UX allows multiple processes to perform operations on the same stream.
- The STREAMS/UX scheduler synchronizes the running of service routines with application processing.

STREAMS/UX Internal Synchronization

STREAMS/UX protects its internal data structures, such as message queues, against interrupts. STREAMS/UX programmers must use the following guidelines.

1 A *put*, *service*, *open*, or *close* routine can pass its own queue or the other queue in its queue pair to a STREAMS/UX kernel utility. Many STREAMS/UX utilities operate on a queue. For example, *getq* takes a queue as an input parameter and returns a message from the queue. A service routine can only pass its queue or the other queue in its queue pair to *getq*. The restricted utilities are *backq, bcanputnext, canputnext, flushband, flushq, freezestr, getq, insq, putbq, putnext, putnextctl, putnextctl1, putnextctl2, putq, qreply, qsize, rmvq, SAMESTR, strqget, strqset,* and *unfreezestr.* The *putq* utility is not restricted when it is passed a driver's read queue or a lower mux's write queue. Any *put* or *service* routine can call *putq* if it passes it a driver's read queue or a lower mux's write queue. However, *putq*'s caller must guarantee that the queue passed is still allocated. Some STREAMS/UX utilities, such as *canput*, are commonly passed a parameter of the form *q->q_next*. These routines are restricted in a slightly different way

than those listed above. A put or service routine can only pass its own queue's *q_nex*t field or the *q_next* field of the other queue in its queue pair. These requirements apply to *bcanput, canput, put, putctl, putctl1, putctl2,* and *streams_put*. These utilities are not restricted when they are passed a parameter of the form *q*, except that the queue must still be allocated.

2 Some STREAMS/UX utilities cannot be called from user functions passed to timeout and bufcall or from non-STREAMS/UX code in the kernel. Also, this software cannot share data structures with STREAMS/UX modules and drivers, unless it raises the *spl* level to protect against interrupts. The utilities which cannot be called are *backq, bcanputnext, canputnext, flushband, flushq, freezestr, getq, insq, putbq, putnext, putnextctl, putnextctl1, putnextctl2, qreply, qsize, rmvq, SAMESTR, strqget, strqset,* and *unfreezestr*. The user functions and non-STREAMS/UX code cannot call *bcanput, canput, put, putctl1, putctl2,* or *streams_put* if they pass the utility a parameter of the form *q->q_next*. They can call these utilities if they pass a parameter of the form *q* (q must be a valid, allocated queue). User functions and non-STREAMS/UX code can only call putq if they pass it a driver's read queue or a lower mux's write queue. User functions and non-STREAMS/UX code can use the new *streams_put* utility documented in this chapter to get around these restrictions.

3 Some STREAMS/UX utilities cannot be called from free routines passed to *esballoc*. A free routine can call the same utilities as the module or driver entry point that called *freeb*.

4 If a multiplexor can execute on the ICS, take care when using *putnext* to pass messages across the multiplexor. If the upper mux passes messages downward by passing the lower mux's write queue to *putnext*, the upper mux must ensure that the driver stays linked under the mux until after the *putnext* completes. Likewise, if the lower mux passes messages upward by passing the upper mux's read queue to putnext, the lower mux must guarantee that the driver stays linked under the mux, the mux stays open, and modules are not pushed or popped until after the *putnext* finishes.

5 A *protect_q* parameter can be passed to the *weldq* utility. The *protect_q* parameter specifies which queue the *func* parameter can access safely. The *func* function can use the same STREAMS/UX utilities as the *protect_q put* and *service* routines.

6 The *put* and *service* routines cannot be called directly. They must be executed by calling STREAMS/UX utilities such as *putnext, putq* or *qenable*. They cannot be called using the function pointer stored in the *q_qinfo* structure.

7 Drivers and modules should not call STREAMS/UX utilities from software running on the interrupt control stack processing an *spl6* or higher interrupt. STREAMS/UX protects its internal data structures using *spl5*.

Driver and Module Synchronization

Drivers and modules must protect their private data structures against interrupts. This can be done in four ways. One way would occur if software that is running on the interrupt control stack (ICS) modifies driver and module data structures. In this case, the driver and module service and put routines must raise the *spl* level before accessing their data structures. Drivers and modules can call the STREAMS/UX utility *splstr* to raise the *spl* level to *spl5*. Interrupts are masked while the *spl* level is raised.

The second way to protect data structures against interrupts is for software running on the ICS to send a message to a stream. If this is done, drivers and modules do not need to raise the *spl* level to protect their data. The software running on the ICS does a *putq* on the driver's read queue. The STREAMS scheduler will run the service routine off the ICS. When ICS software calls *putq* for a priority band, the driver open function must allocate the band by calling *strqget*. This prevents *putq* from dynamically allocating memory for the band on the ICS.

ICS software can call *putnext* or *put* instead of *putq* to send a message to a stream. If one of these utilities is called, STREAMS/UX will attempt to run the put routine on the ICS. Drivers and modules will need to use *spl* calls to protect data structures that they share with other drivers and modules, with other instances of the same driver or module, or with non-STREAMS/UX software.

The third way to protect data structures against interrupts is for interrupt software to call the *qenable* utility to schedule a service routine. The STREAMS/UX scheduler will run the service routine off the ICS.

The fourth method for protecting data structures against interrupts is to call the new *streams_put* utility. The code running on the ICS passes streams_put a function and a queue. STREAMS/UX runs the function as if it were the queue's put routine. The function can access the same data structures as the queue's put routine. See “HP-UX Modifications to STREAMS/UX Utilities” in this chapter for more information about *streams_put*.

Multiple Processes Accessing the Same Stream

STREAMS/UX synchronizes multiple processes that are accessing the same stream. Three scenarios will allow more than one process to operate on a stream:

- Multiple processes opening a non-cloneable device with the same minor number
- A process calling fork
- Processes issuing I_SENDFD and I_RECVFD ioctls

For synchronization, STREAMS/UX will queue some *open* and *ioctl* system calls issued by different processes, and will execute them one at a time. STREAMS/UX queues re-opening an already open stream, and queues the following ioctls: I_PUSH, I_POP, I_LINK, I_PLINK, I_UNLINK, I_PUNLINK, I_FLUSH, I_FLUSHBAND, I_GETCLTIME, I_SETCLTIME, I_GETSIG, I_SETSIG, I_LIST, I_LOOK, and I_STR.

STREAMS/UX does not process a *close* call until the last file descriptor for a stream is closed. No other system calls will be executing when STREAMS/UX begins to dismantle the stream.

For remaining system calls, STREAMS/UX ensures that consistent results are returned, but the calls are not executed one at a time. For example, if two processes are reading from the same stream, one process could read the first and third messages on the stream to satisfy a read request while the second process reads the second and fourth messages.

The STREAMS/UX Scheduler

The STREAMS/UX scheduler runs service routines that are scheduled by STREAMS/UX utilities such as *putq*. The scheduler will run all scheduled service routines before returning to user level. The scheduler is a real time daemon that runs at priority 100. (A low priority number denotes a high priority. For example, a priority number of 50 would be of higher priority than the number 100.) STREAMS/UX applications need to run at a lower priority (higher priority number) than the STREAMS/UX scheduler; otherwise service routines will not run before the scheduler returns to user level from the kernel.

HP-UX Changes to Cloning

STREAMS/UX supports two methods of cloning. See the SVR4.2 STREAMS manual for more information about cloning. Some differences exist between HP-UX cloning and SVR4.2 cloning.

The first cloning method uses a special clone major number, 72, to provide cloning. For each cloneable device, a device file must exist that has the clone major number of 72 and also has a minor number equal to the major number of the real device. When an application opens this type of device file, STREAMS/UX passes the driver open routine CLONEOPEN in the *sflag* parameter. The driver allocates a minor number and returns a new device number containing the true major number and the chosen minor number. The driver uses either *makdev* or to create the new device number.

The second cloning method is useful for drivers which need to be able to encode information in their minor numbers. This is not possible in the first method, as the clone device file for that method must have as its minor number the major number of the driver being cloned.

In the second cloning method, the driver designates a particular minor number as its “clone” minor number. The driver open routine checks the minor number portion of the device number parameter passed to it, and if it is the clone minor number, the driver open routine allocates a minor number and returns a new device number to the caller, in the same way as the first cloning method described above. The returned device number must contain both a major number and the new minor number. A driver using this cloning method may also change the major number in the device number it returns. However, the new major number must correspond to a STREAMS/UX driver with the same streamtab structure as the driver associated with the original major number. Also, on a multiprocessor system, if the original driver was MP scalable, the new one must be too. Likewise, if the original was UP emulation, the new one must be also.

Drivers using the second cloning method must indicate this in their install functions or master file entries. See Chapter 5 for more information about configuring STREAMS/UX drivers. Install functions must set the C_CLONESMAJOR flag. For example:

INSTALL FUNCTION CONFIGURATION

```
static drv_info_t example_drv_info = {    /*driver information*/
       “example”,                         /* name */
       “pseudo”,                          /* class */
       DRV_CHAR | DRV_PSEUDO,              /* flags */
       -1,                                /* block major number */
       -1,                               /* dynamically assigned
                                             character major number */
       NULL, NULL, NULL,                 /* cdio, gio_private,and
                                             cdio_private structures */
}

static drv_ops_t example_drv_ops = {      /* driver entry points */
       NULL,                             /* open */
       NULL,                             /* close */
       NULL,                             /* strategy */
       NULL,                             /* dump */
       NULL,                             /* psize */
       NULL,                             /* mount */
       NULL,                             /* read */
       NULL,                             /* write */
       NULL,                             /* ioctl */
       NULL,                             /* select */
       NULL,                             /* option1 */
       NULL, NULL, NULL, NULL,           /* reserved entry points */
       C_CLONESMAJOR,                    /* ****NOTE****C_CLONESMAJOR
flag set */
     }

static streams_info_t example_str_info = { /* streams information */
       example,                            /* name */
       -1,                                 /* dynamically assigned major
                                              number */
       { &examplerinit, &examplewinit,
         NULL,NULL },                      /* streamtab */
       STR_IS_DEVICE,                      /* flags */
       0,                                  /* synchronization level */
       “”,                                 /* elsewhere sync name */
}

int
example_install()
{
       int retval;
```

MASTER FILE ENTRY

```
$DRIVER_INSTALL
* Driver          Block major          Char major
example              1                    -1


if ((retval = install_driver(&example_drv_info, &example_drv_ops))!= 0)
        return(retval);

/* Configure streams specific parameters. */
if ((retval = str_install(&example_str_info)) != 0) {
   uninstall_driver(&example_drv_info);
   return(retval);
   }

   /* Success */
   return 0;

}
```

For definition in the $DEVICE table in the driver's master file entry, set the 0x8000 bit in the mask field to use the second cloning method. For example:

MASTER FILE $DEVICE TABLE CONFIGURATION

```
name     handle       type  mask   block  char

example  exampleinfo  21    80FC   -1     75  /* 0x8000 set in mask */
```

STREAMS/UX Hardware Driver Writing

STREAMS/UX does not provide all the kernel utilities needed to write a hardware driver. STREAMS/UX provides only the utilities described in this manual. Customers who need to write hardware drivers should contact their HP representative for additional support.

4

STREAMS/UX Multiprocessor Support

This chapter describes how STREAMS/UX runs on a multiprocessor (MP) system. The following topics are covered:

- How to run modules and drivers in uniprocessor (UP) emulation mode.
- How to write MP scalable modules and drivers.
- How to port SVR4 MP modules and drivers to HP-UX.
- How to use MP synchronization levels on a uniprocessor system to protect against interrupts.

Running Modules and Drivers in Uniprocessor Emulation Mode

STREAMS/UX supports uniprocessor emulation for modules and drivers. Modules and drivers which run on uniprocessor systems can run on multiprocessor systems under UP emulation without code changes. This section presents an overview of UP emulation, describes how to configure modules and drivers for UP emulation, describes what happens when a stream contains both UP emulation and MP scalable modules, and describes how UP emulation affects performance. Lastly, this section contains some UP emulation programming guidelines.

How STREAMS/UX Executes UP Emulation Modules and Drivers

This section describes how STREAMS/UX supports UP emulation. HP-UX provides UP emulation for non-STREAMS device drivers which were developed for uniprocessor systems.

HP-UX uses a semaphore called the I/O semaphore and a spinlock known as the *spl* lock to implement UP emulation. HP-UX uses the I/O semaphore to serialize driver system calls. HP-UX acquires the I/O semaphore before calling the driver to process a system call. HP-UX uses the *spl* lock to prevent a driver interrupt on one processor from running in parallel with a driver system call on another processor. When a driver calls *spln* to raise the *spl* level, HP-UX acquires the *spl* lock. When an interrupt occurs for a UP emulation driver, HP-UX acquires the *spl* lock before calling the driver's interrupt handler.

STREAMS/UX extends UP emulation for STREAMS/UX modules and drivers. A stream can be entered in two ways. One way is through a system call. Either the I/O system or STREAMS/UX acquires the I/O semaphore before executing a system call for a UP emulation stream. Also, a stream can be entered from non-STREAMS software in the kernel. For example, an interrupt handler can call *putq, putnext,put,* or *streams_put* to enter a stream.

If code on the interrupt control stack (ICS) calls *putq* for a UP emulation stream, the STREAMS/UX UP emulation scheduler runs the service routine. This scheduler acquires the I/O semaphore. If an interrupt occurs for a UP

emulation driver, the I/O system acquires the *spl* lock. Then if the interrupt handler calls *put*, *putnext*, or *streams_put*, STREAMS/UX usually executes the put routine on the ICS with the spl lock. Note that the STREAMS/UX utilities do not acquire the *spl* lock. An MP scalable interrupt handler may not be able to safely call *put*, *putnext*, or *streams_put* to enter a UP emulation stream.

STREAMS/UX protects the various callback functions in different ways. STREAMS/UX does not have to acquire the I/O semaphore or *spl* lock to run *esballoc* free routines in UP emulation mode. The free routine will automatically run in the same mode as the module which calls *freeb*. Also, the HP-UX I/O system protects timeout callback routines by obtaining the *spl* lock before running the routine.

Bufcall and *weld* callback functions are always run under UP emulation. The STREAMS/UX memory and weld daemons always obtain the I/O semaphore before running UP emulation or MP scalable callback routines. This should not hurt the performance of MP scalable modules because *weldq* and *bufcall* are not called very often.

Configuring Modules and Drivers for UP Emulation

Modules and drivers run in UP emulation mode by default. To configure a module or driver to run in UP emulation mode, do not specify any MP flags. The examples below show how to configure UP emulation modules and drivers by creating a master file $DEVICE table entry or a module or driver install function. See Chapter 5 for more information about configuring modules and drivers.

MASTER FILE $DEVICE TABLE CONFIGURATION

```
name         handle       type   mask  block   char

lo           loinfo       21      FC    -1     75   /* 0x10000 not set in mask */
lmodb        lmbinfo      40      0     -1     -1   /* 0x10000 not set in mask */
```

INSTALL FUNCTION CONFIGURATION

LO DRIVER

```
static drv_info_t lo_drv_info = {    /* driver information */
      “lo”,                          /* name */
      “pseudo”,                      /* class */
      DRV_CHAR | DRV_PSEUDO,         /* *****NOTE***** DRV_MP_SAFE flag not specified */
      -1,                            /* block major number */
      75,                            /* character major number */
      NULL, NULL, NULL,              /* cdio, gio_private, and cdio_private structures */
 }


static drv_ops_t lo_drv_ops = {       /* driver entry points */
      NULL,                           /* open */
      NULL,                           /* close */
      NULL,                           /* strategy */
      NULL,                           /* dump */
      NULL,                           /* psize */
      NULL,                           /* mount */
      NULL,                           /* read */
      NULL,                           /* write */
      NULL,                           /* ioctl */
      NULL,                           /* select */
      NULL,                           /* option1 */
      NULL, NULL, NULL, NULL,         /* reserved entry points */
      0,                              /* device flags */
}

static streams_info_t lo_str_info = {   /* streams information */
      “lo”,                           /* name */
      75,                             /* major number */
      {&lorinit, &lowinit, NULL, NULL},/* streamtab */
      STR_IS_DEVICE,                  /* *****NOTE***** MGR_IS_MP flag not specified */
      0,                              /* synchronization level */
      “”,                             /* elsewhere sync name */
}
```

```
int
lo_install()
{
   int retval;

   if ((retval = install_driver(&lo_drv_info, &lo_drv_ops)) != 0)
      return(retval);

   if ((retval = str_install(&lo_str_info)) != 0) {
      uninstall_driver(&lo_drv_info);
      return(retval);
   }

   /* success */
   return 0;

}
LMODB MODULE

static streams_info_t lmodb_str_info = { /* streams information */
     “lmodb”,                            /* name */
     -1,                                 /* major number */
     { &lmodbrinit, &lmodbwinit },       /* streamtab */
     STR_IS_MODULE,                      /* *****NOTE***** MGR_IS_MP flag not
                                            specified */
     0,                                  /* synchronization level */
     “”,                                 /* elsewhere sync name */
}

int
lmodb_install()
{
   int retval;

   return(str_install(&lmodb_str_info));

     }
```

Mixing MP Scalable and UP Emulation Modules and Drivers

Because UP emulation and MP scalability are configured separately for each module or driver, it is possible for a stream to contain both UP emulation and MP scalable modules and drivers. If any module or driver in a stream needs to run in UP emulation mode, STREAMS/UX runs the entire stream under UP emulation.

When a module is pushed onto a stream, STREAMS/UX checks if either the module is configured for UP emulation or if the stream is running under UP emulation. If either condition is true, the module and the entire stream run under UP emulation. Also, when the module is popped, the stream does not change back to its original mode.

When a driver is linked under a multiplexor, STREAMS/UX checks if both streams run in the same mode. If they do not, STREAMS/UX changes the MP scalable stream to run in UP emulation mode. When the driver is unlinked, STREAMS/UX does not change a stream back to its original mode.

STREAMS/UX does not support mixing MP scalable and UP emulation modules in an upper mux because an upper mux is a clonable device. STREAMS/UX does not detect that upper mux streams are related. In particular, STREAMS/UX does not support pushing a UP emulation module onto only one MP scalable upper mux stream. STREAMS/UX changes only this one stream to run under UP emulation. It does not change the control stream or the other upper mux streams. You should design your modules and drivers so that only MP scalable modules are pushed onto MP scalable upper muxes. Also, STREAMS/UX does not support linking a UP emulation driver under an MP scalable upper mux. STREAMS/UX only changes the control stream to run under UP emulation. It does not change the other upper mux streams. You should link only MP scalable drivers under an MP scalable upper mux.

Some examples of supported streams configurations which contain both MP scalable and UP emulation modules and drivers are listed below.

- If an MP scalable driver is linked under a UP emulation mux, STREAMS/UX changes the MP scalable driver to run in UP emulation mode. For example, DLPI is MP scalable in 10.0. When it is linked under UP emulation SNA, STREAMS/UX changes the drivers to run in UP emulation mode.
- When an MP scalable module is pushed onto a UP emulation stream, STREAMS/UX runs the module under UP emulation. For example, timod is MP scalable. When it is pushed onto a UP emulation OSI stream, it runs under UP emulation.
- When a UP emulation module is pushed onto an MP scalable stream, STREAMS/UX changes the entire stream to run under UP emulation. For example, DLPI is MP scalable in Release 10.0. When UP emulation Portable Netware modules are pushed onto DLPI, the entire stream runs in UP emulation

mode. Another example is STREAMS/UX pipes, which are MP scalable. If UP emulation modules are pushed onto a pipe, the pipe runs under UP emulation.

- As described earlier in this section, all user *bufcall* callback functions are executed in UP emulation mode. If an MP scalable module calls *bufcall*, the callback routine runs under UP emulation. If the callback routine invokes a put procedure, the put procedure also runs in UP emulation mode. For example, the DLPI driver is MP scalable in 10.0 and calls *bufcall*. The *bufcall* callback function runs under UP emulation.

Performance

Performance of UP emulation modules and drivers will likely worsen as more processors are added to a system. If a large number of users will be running your modules and drivers on MP systems, you should probably modify the code to be MP scalable.

MP scalable modules that run over non-STREAMS/UX UP emulation drivers will be forced to run in UP emulation mode. You can achieve better performance by changing drivers to be MP scalable.

Guidelines for UP Emulation Modules and Drivers

- It is easier to develop STREAMS/UX based software which runs completely under UP emulation or is completely MP scalable. Try to avoid mixing UP emulation and MP scalable modules and drivers in the same stream or multiplexor.
- It may be safe for UP emulation modules and drivers to call MP scalable non-STREAMS software. The MP scalable software must be able to run while the I/O semaphore is held. Note that if a put or service routine calls non-STREAMS functions, these functions cannot acquire semaphores because this might cause the *put* or *service* routine to block.
- Be careful with MP scalable non-STREAMS kernel code when calling UP emulation STREAMS/UX modules. It is better if the non-STREAMS code schedules a service routine instead of invoking a put procedure. Scheduling the service routine will wake up the UP emulation scheduler daemon to run the routine. The daemon acquires the I/O semaphore. If non-STREAMS code calls *put*, *putnext*, or *streams_put*, STREAMS/UX will not acquire either the I/O semaphore or the *spl* lock.
- Modules and drivers which can run MP scalable and run under UP emulation must use queue or queue pair synchronization. An example of an MP scalable

module which can run in UP emulation mode is timod. Although timod will be configured to be MP scalable, it is pushed onto many streams, some of which run in UP emulation mode.

- Do not push a UP emulation module onto an MP scalable upper mux. Do not link a UP emulation driver under an MP scalable upper mux. It is better for the mux to contain either all MP scalable modules and drivers or all UP emulation modules and drivers.
- The UP emulation scheduler runs differently from the uniprocessor scheduler. This may affect STREAMS application programs. On multiprocessor systems, the scheduler may not run a service routine before the process which scheduled the routine returns to user level.
- UP emulation modules and drivers need to follow the guidelines in the "STREAMS/UX Uniprocessor Synchronization" section of Chapter 3.

Writing MP Scalable Modules and Drivers

Overview of STREAMS/UX MP Support

HP-UX STREAMS supports MP scalable drivers and modules. You can configure the amount of parallelism for modules and drivers. Pick a level which is consistent with a module's or driver's use of shared data structures. STREAMS/UX provides five levels of parallelism which are called *queue, queue pair, module*, *elsewhere*, and *global*. They are described below. Also, STREAMS provides extra synchronization for module and driver *open* and *close* functions. This synchronization is also described below. The term module is used in this discussion to mean both modules and drivers, unless otherwise stated.

Figure 1 is useful for understanding STREAMS/UX MP support. The diagram shows four streams, ECHO-A, ECHO-B, DLPI-A and SAD-A. ECHO-A and ECHO-B both contain the *echo* driver. DLPI-A contains *dlpi*, and SAD-A has *sad*. Each driver contains a read and a write queue. *echo_rput* and *echo_rsrv* operate on an echo driver's read queue. *echo_wput* and *echo_wsrv* access the write queue. The *dlpi* and *sad* driver functions are similar to the echo driver functions. STREAMS/UX executes *echo, dlpi*, and *sad* driver functions differently depending on the MP synchronization level configured for the drivers.

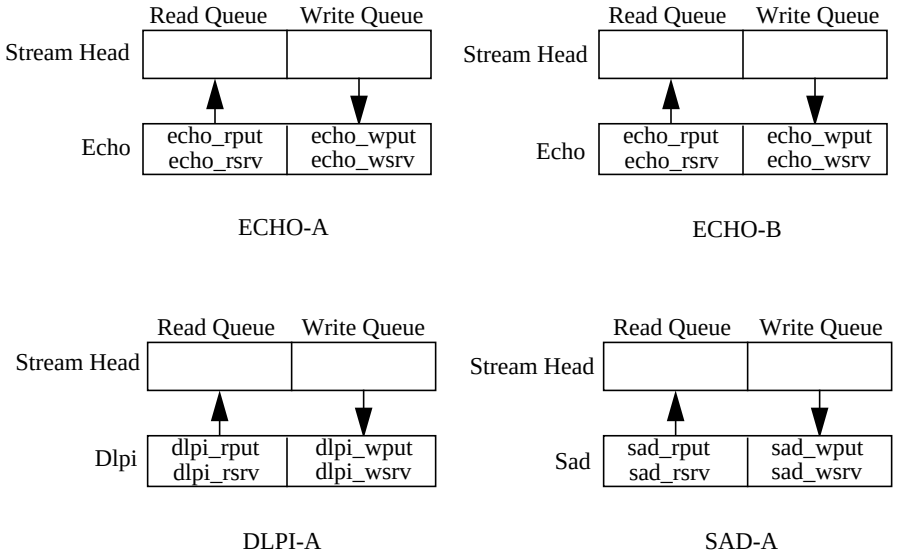


Figure 1 **Understanding STREAMS/UX MP Support**

The queue synchronization level provides the most concurrency. It serializes access to a queue so that only one function at a time can manipulate the queue. Applications can take advantage of multiple processors because functions that operate on different queues run in parallel. For example, assume that the echo driver in Figure 1 uses queue synchronization. STREAMS/UX does not run ECHO-A's *echo_rput* and echo_rsrv in parallel. Also, STREAMS/UX does not execute ECHO-A's *echo_wput* and *echo_wsrv* concurrently. However, STREAMS/UX can run ECHO-A's *echo_rput* at the same time as ECHO-A's *echo_wput*. STREAMS/UX allows ECHO-A's read queue functions to run in parallel with ECHO-A's write queue routines. Also, any of ECHO-A's procedures can run at the same time as ECHO-B, DLPI-A or SAD-A routines. If a module uses queue synchronization, a queue's put and service routines can easily share data with each other because STREAMS/UX does not execute the routines concurrently.

The *queue pair* synchronization level serializes access to a read and write queue pair so that only one of the queue pair's functions can run at a time. Queue pair synchronization still allows concurrency because functions for

different queue pairs run in parallel. (A queue pair is also known as a module instance.) For example, assume that the echo driver in Figure 1 is configured for queue pair synchronization. STREAMS/UX runs ECHO-A's *echo_rput*, *echo_rsrv*, *echo_wput*, and *echo_wsrv* one at a time. In other words, STREAMS/UX does not execute any of ECHO-A's echo driver functions concurrently, nor will STREAMS/UX run any of ECHO-B's echo driver functions in parallel. However, STREAMS/UX can run an ECHO-A function at the same time as an ECHO-B function. Also, any of ECHO-A's functions can run in parallel with DLPI-A or SAD-A routines. If a module uses queue pair synchronization, a queue pair's functions run one at a time and can share data.

The *module* synchronization level serializes access to all of a module's queue pairs or instances. STREAMS/UX runs only one function at a time for all of a module's queue pairs. However, STREAMS/UX runs functions for different modules in parallel. Modules are different if they have different master file entries. For example, timod and tirdwr are different modules. Assume that the echo driver in Figure 1 is configured for module synchronization. STREAMS/UX does not run *echo* driver functions in ECHO-A and ECHO-B in parallel.

However, STREAMS/UX can run an echo driver function at the same time as dlpi or a sad driver function. Because STREAMS/UX allows only one function for all of a module's queue pairs to run at a time, the module's queue pairs can share data.

The *elsewhere* synchronization level serializes a group of different modules. STREAMS/UX runs only one function at a time for the group of modules. STREAMS/UX runs functions in different groups concurrently. Suppose the echo and dlpi drivers in Figure 1 are configured to be members of an *elsewhere* synchronization group. Also, assume the sad driver is configured to be in a different *elsewhere* group. Only one driver function in ECHO-A, ECHO-B and DLPI-A can run at a time. However, a function in ECHO-A, ECHO-B or DLPI-A can run in parallel with a function in SAD-A. Also, a function in ECHO-A, ECHO-B or DLPI-A can run at the same time as a function in a module which uses a different synchronization level than *elsewhere*. The modules in a group can share data.

The *global* synchronization level does not provide parallelism within STREAMS/UX. Only one module out of those configured for global synchronization can run at a time. Suppose that in Figure 1, the echo, dlpi

and sad drivers use global synchronization. Only one driver function in ECHO-A, ECHO-B, DLPI-A and SAD-A can run at a time. However, one of these drivers could run in parallel with a module configured for a synchronization level other than *global*. All modules configured with *global* synchronization can share data.

The STREAMS/UX synchronization levels also apply to open and close. For example, if a module is configured for queue pair synchronization, none of the *put* or *service* routines for the queue pair can run at the same time as the queue pair's *open* or *close*. Also, open cannot run at the same time as close. The least amount of protection that STREAMS/UX provides for opens and closes is queue pair. Even if a module is configured with queue synchronization, it will run as if it were configured with queue pair synchronization during opens and closes.

STREAMS/UX provides additional protection for *opens* and *closes*. STREAMS/UX executes only one *open* or *close* across all streams at a time. For example in Figure 1, if STREAMS/UX is executing the ECHO-A echo driver's *open* routine, the DLPI-A dlpi *open* cannot run nor can any other module's or driver's *open* or *close*. An exception to this occurs if an *open* or *close* sleeps. When this happens, other *opens* and *closes* can occur. An *open* or *close* function that sleeps may need to use a spinlock together with the *get_sleep_lock*, SV_WAIT or SV_WAIT_SIG utilities to prevent missing wakeups. These utilities are described in the “HP-UX Modifications to STREAMS/UX Utilities” section in Chapter 3. Also, SV_WAIT and SV_WAIT_SIG are discussed in the SVR4.2 Driver manual.

STREAMS does not synchronize the running of timeout and bufcall callback functions with modules and drivers. This chapter lists some restrictions on what these callback functions can do.

Suggestions for Designing MP Scalable Modules and Drivers

This section contains recommendations for designing MP scalable modules and drivers:

- Modules and drivers that run over UP emulation hardware drivers must run under UP emulation. Before changing STREAMS/UX modules and drivers to be MP scalable, modify hardware drivers to be MP scalable.
- You can improve the performance of modules and drivers by using the elsewhere synchronization level. Configure all modules and drivers in a subsystem to be in

the same group. They can all share data. However, STREAMS/UX will not synchronize *bufcall* and *timeout* callback functions or any non-STREAMS/UX code with the modules or drivers. You may be able to use the *streams_put* utility described in Chapter 3. In general, UP emulation provides more protection for *bufcall*, *timeout*, and non-STREAMS functions.

- To change modules and drivers to be MP scalable, analyze how the code shares data structures. Determine which structures are shared and which module and driver entry points read and write to the structures. Using this information, choose synchronization levels for modules and drivers that correctly serialize access to shared data.
- If all modules and drivers of a product share the same structure, consider changing the module and driver data structures and algorithms to allow for more parallelism. Alternatively, consider using spinlocks to protect shared structures that are accessed infrequently or for short amounts of time. Using spinlocks is a good way to protect structures which are not accessed on the main read and write paths. You can either use the native HP-UX spinlock primitives or the SVR4 MP *LOCK*, *TRYLOCK*, *UNLOCK*, *LOCK_ALLOC* and *LOCK_DEALLOC* utilities. The SVR4 MP utilities are discussed under "HP-UX Modifications to STREAMS/UX Utilities" in Chapter 3 and in the SVR4.2 Driver manual.
- Use service routines only for flow control, recovering from resource shortages or executing interrupt completions in a process context. Service routines degrade performance.
- Be careful when writing *timeout* and *bufcall* callback functions, as well as non-STREAMS code that calls STREAMS/UX utilities or shares data with modules and drivers. See the "Guidelines for MP Scalable Modules and Drivers" section.

Configuring MP Scalable Modules and Drivers

This section describes how to configure MP scalable modules and drivers.

MP Scalable Module and Driver Configuration

If you want a module or driver to be MP scalable, you must specify additional configuration parameters. You need to:

- Add a flag indicating that the module or driver is MP scalable
- Add a keyword which specifies the synchronization level the module or driver uses
- Add a *sync* name if the module or driver requires *elsewhere* synchronization

The sync name indicates which modules and drivers belong to a group. Choose a sync name with eight characters or less, and configure the name for each member of the group. See Chapter 5 for more information about configuring STREAMS/UX modules and drivers.

Master File $DEVICE Table Configuration

To configure an MP scalable module or driver using a master file $DEVICE table entry, add the 0x10000 (MGR_IS_MP) flag to the mask value. Also add an entry to the master file $STREAMS_DVR_SYNC table. This entry contains the module or driver's name, a keyword specifying the synchronization level, and a *sync* name if the module or driver requires elsewhere synchronization. There are five synchronization level keywords: *sync_global*, *sync_elsewhere*, *sync_module*, *sync_qpair*, and *sync_queue*. The STREAMS/UX master file contains a list of the valid keywords in the $STREAMS_SYNC_LEVEL table. The examples below show $DEVICE and $STREAMS_DVR_SYNC table entries.

```
* name       handle        type    mask   block   char
*
$DEVICE
strlog       loginfo       21     120FC   -1     73  /* Added 0x10000 to mask */
dlpi         dlpiinfo      21     120FC   -1     119 /* Added 0x10000 to mask */
tirdwr       tirdwrinfo     40     12000   -1     -1  /* Added 0x10000 to mask */
A            Ainfo         40     12000   -1     -1  /* Added 0x10000 to mask */
B            Binfo         40     12000   -1     -1  /* Added 0x10000 to mask */
C            Cinfo         40     12000   -1     -1  /* Added 0x10000 to mask */
D            Dinfo         21     120FC   -1     116 /* Added 0x10000 to mask */
$$$
* name         sync level       sync name
*
$STREAMS_DVR_SYNC
strlog         sync_module                      /* Added sync level */
dlpi           sync_qpair                       /* Added sync level */
tirdwr         sync_queue                       /* Added sync level */
A              sync_elsewhere   ABsync            /* Added sync level & name
*/
B              sync_elsewhere   ABsync           /* Added sync level & name */
C              sync_elsewhere   netsync          /* Added sync level & name */
D              sync_elsewhere   netsync          /* Added sync level & name */
$$$
```

Module and Driver Install Function Configuration

If a module or driver is configured using an install function, add the MGR_IS_MP flag to the *inst_flags* field in the *streams_info_t* structure. Also, if you are configuring a driver, set the DRV_MP_SAFE flag in the

drv_info_t structure. Specify a synchronization level in the *inst_sync_level* field. The possible values are *SQLVL_GLOBAL*, *SQLVL_ELSEWHERE*, *SQLVL_MODULE*, *SQLVL_QUEUEPAIR* and *SQLVL_QUEUE*. If the module or driver is using the elsewhere synchronization level, add a sync name to the *inst_sync_info* field. Note that a module or driver which uses an install function for configuration needs an entry in the master file $DRIVER_INSTALL table. (Do not put an entry in the $DEVICE table if an install function is used.) The examples below show MP scalable module and driver install functions.

```
STRLOG DRIVER

static drv_info_t strlog_drv_info = {     /* driver information */
      “strlog”,                           /* name */
      “pseudo”,                           /* class */
      DRV_CHAR | DRV_PSEUDO |             /* *****NOTE***** DRV_MP_SAFE flag specified */
      DRV_MP_SAFE,
      -1,                                 /* block major number */
      73,                                 /* character major number */
      NULL, NULL, Null,                   /* cdio, gio_private, and cdio_private
                                             structures

}

static drv_ops_t strlog_drv_ops = {       /* driver entry points */
      NULL,                               /* open */
      NULL,                               /* close */
      NULL,                               /* strategy */
      NULL,                               /* dump */
      NULL,                               /* psize */
      NULL,                               /* mount */
      NULL,                               /* read */
      NULL,                               /* write */
      NULL,                               /* ioctl */
      NULL,                               /* select */
      NULL,                               /* option1 */
      NULL, NULL, NULL, NULL,             /* reserved entry points */
      0,                                  /* device flags */
}
static streams_info_t strlog_str_info = { /* streams information */
      “strlog”,                           /* name */
      73,                                 /* major number */
      {&loginit, &logwinit, NULL, NULL},  /* streamtab */
      STR_IS_DEVICE | STR_SYSV4_OPEN |    /* *****NOTE***** MGR_IS_MP flag specified */
      MGR_IS_MP,
      SQLVL_MODULE,                       /* *****NOTE***** synch level specified */
      “”,                                 /* elsewhere sync name */

  }


}
```

```
int
strlog_install()
{
   int retval;

   if ((retval = install_driver(&strlog_drv_info, &strlog_drv_ops)) != 0)
      return(retval);

   if ((retval = str_install(&strlog_str_info)) != 0) {
      uninstall_driver(&strlog_drv_info);
      return(retval);
   }

   /* success */
   return 0;
```

TIRDWR MODULE

```
static streams_info_t tirdwr_str_info = { /* streams information */
     “tirdwr”,                            /* name */
     -1,                                  /* major number */
     { &rinit, &winit, NULL, NULL },      /* streamtab */
     STR_IS_MODULE | STR_SYSV4_OPEN |     /* *****NOTE***** MGR_IS_MP flag specified */
     MGR_IS_MP
     SQLVL_QUEUE,                         /* *****NOTE***** synch level specified */
     “”,                                  /* elsewhere sync name */
}

int
tirdwr_install()
{
   int retval;

   return(str_install(&tirdwr_str_info));
}
```

C MODULE

```
static streams_info_t c_str_info = {      /* streams information */
     “C”,                                 /* name */
     -1,                                  /* major number */
     { &crinit, &cwinit, NULL, NULL },    /* streamtab */
     STR_IS_MODULE | STR_SYSV4_OPEN |     /* *****NOTE***** MGR_IS_MP flag specified
     MGR_IS_MP
     SQLVL_ELSEWHERE,                     /* *****NOTE***** synch level specified */
     “netsync”,                           /* *****NOTE***** sync name specified */
}
```

```
int
C_install()
{
   int retval;

   return(str_install(&c_str_info));

}

D DRIVER

static drv_info_t d_drv_info = {        /* driver information */
      “D”,                              /* name */
      “pseudo”,                         /* class */
      DRV_CHAR | DRV_PSEUDO |           /* *****NOTE***** DRV_MP_SAFE flag specified */
      DRV_MP_SAFE,
      -1,                               /* block major number */
      -1,                               /* dynamically assigned character major number */
     NULL, NULL, NULL,                  /* cdio, gio_private, and cdio_private
                                           structures */
}

static drv_ops_t d_drv_ops = {          /* driver entry points */
      NULL,                             /* open */
      NULL,                             /* close */
      NULL,                             /* strategy */
      NULL,                             /* dump */
      NULL,                             /* psize */
      NULL,                             /* mount */
      NULL,                             /* read */
      NULL,                             /* write */
      NULL,                             /* ioctl */
      NULL,                             /* select */
      NULL,                             /* option1 */
      NULL, NULL, NULL, NULL,           /* reserved entry points */
      0,                                /* device flags */
}

static streams_info_t d_str_info =  {  /* streams information */
      “D”,                             /* name */
      -1,                              /* dynamically assigned major number */
      { &drinit, &dwinit, NULL, NULL}, /* streamtab */
      STR_IS_DEVICE | STR_SYSV4_OPEN | /* *****NOTE***** MGR_IS_MP flag specified */
      MGR_IS_MP,
      SQLVL_ELSEWHERE,                 /* *****NOTE***** synch level specified */
      “netsync”,                       /* *****NOTE***** sync name specified */
}
```

```
int
D_install()
{
   int retval;

   /* Configure driver and obtain dynamically assigned major number. */
   if ((retval = install_driver(&d_drv_info, &d_drv_ops)) != 0)
      return(retval);


   /* Configure streams specific parameters. */
   if ((retval = str_install(&d_str_info)) != 0) {
      uninstall_driver(&d_drv_info);
      return(retval);
   }

   /* Success */
   return 0;

}
```

Configuring the NSTRSCHED Tunable

STREAMS/UX provides a new tunable, NSTRSCHED, which allows you to set the number of STREAMS/UX scheduler daemons running on a multiprocessor system. The default value is 0, which indicates that STREAMS/UX will determine the number of daemons based on the number of processors in the system. The minimum value is 0 and the maximum is 32.

You should leave NSTRSCHED set to the default value. STREAMS/UX will set the number of daemons based on the number of processors in the system. STREAMS/UX will create fewer daemons than there are processors. There is no benefit to creating more daemons than processors. You might want to increase the value of NSTRSCHED if the system does a lot of STREAMS/UX processing or decrease it if the system does very little STREAMS/UX work. You can determine the number of scheduler daemons running on the system by executing the *ps -ef* command, and counting the number of *smpsched* processes.

Guidelines for MP Scalable Modules and Drivers

- It is easier to develop STREAMS/UX-based software that runs completely MP scalable or completely under UP emulation. Try to avoid mixing MP scalable and UP emulation modules and drivers in the same stream or multiplexor.
- MP scalable STREAMS/UX modules and drivers cannot call UP emulation software. A put or service routine cannot acquire the I/O semaphore because *put*

and *service* routines cannot block. This means, for example, that modules and drivers which run over a UP emulation hardware driver must run under UP emulation.

- Modules and drivers which can run both MP scalable and under UP emulation must use queue or queue pair synchronization. An example of an MP scalable module which can run in UP emulation mode is *timod*. Although *timod* is configured to be MP scalable, it is pushed onto many streams, some of which run in UP emulation mode.
- The MP scheduler runs differently from the uniprocessor scheduler. This may affect STREAMS/UX application programs. On multiprocessor systems, the scheduler may not run a service routine before the process which scheduled the routine returns to user level.
- A module or driver's synchronization level determines the entities with which it can share data. It also determines the entities with which it can share its STREAMS/UX queues. For example, if a module uses queue pair synchronization, the write *put* routine can call *insq* to insert a message onto the module's read queue. But, if the module uses queue synchronization, the write put routine can only call *insq* to insert messages onto the write queue. The synchronization level determines which queues a module or driver can pass to STREAMS/UX utilities.

  In general, a *put* or *service* routine can only pass its own queue or queues belonging to entities with which it can share data. The restricted utilities are *backq, bcanputnext, canputnext, flushband, flushq, freezestr, getq, insq, putbq, putnext, putnextctl, putnextctl1, putnextctl2, putq, qreply, qsize, rmvq, SAMESTR, strqget, strqset* and *unfreezestr*. The *putq* utility is not restricted when it is passed a driver's read queue or a lower mux's write queue. Any *put* or *service* routine can call *putq* if it passes a driver's read queue or a lower mux's write queue. However, *putq*'s caller must guarantee that the queue passed in is still allocated.

  Some STREAMS/UX utilities, such as canput, are commonly passed a parameter of the form *q->q_next.* These routines are restricted in a different way from those listed above. A put or service routine can only pass its own queue's *q_nex*t field or the *q_nex*t field of queues belonging to entities with which it can share data. These requirements apply to *bcanput, canput, put, putctl, putctl1, putctl2,* and *streams_put*. These utilities are not restricted when they are passed a parameter of the form *q*, except that the queue must still be allocated.
- Some restrictions exist for timeout and bufcall callback routines as well as non-STREAMS/UX code in the kernel. This software cannot share data

structures with STREAMS/UX modules and drivers, unless spinlocks are used to protect critical sections. Also, the code cannot call the following utilities: *backq, bcanputnext, canputnext, flushband, flushq, freezestr, getq, insq, putbq, putnext, putnextctl, putnextctl1, putnextctl2, qreply, qsize, rmvq, SAMESTR, strqget, strqset*, and *unfreezestr*.

Callback routines and non-STREAMS code cannot call *bcanput, canput, put, putctl, putctl1, putctl2* or *streams_put* if they pass the utility a parameter of the form *q->q_next*. They can call these utilities if they pass a parameter of the form *q* (*q* must be a valid, allocated queue). Callback and non-STREAMS code can call *putq* only if they pass it a driver's read queue or a lower mux's write queue. Callback and non-STREAMS code can use the new *streams_put* utility documented in the section “HP-UX Modifications to STREAMS/UX Utilities” in Chapter 3.

- Some restrictions exist on free routines passed to *esballoc*. A free routine can call STREAMS/UX utilities in the same way as the put or service routine that calls *freeb*. A free routine can access the same data structures as the put or service routine that calls freeb.
- A *protect_q* parameter can be passed to the *weldq* utility. The *protect_q* parameter specifies which queue the *func* parameter can access safely. The *func* function can use the same STREAMS/UX utilities as the *protect_q put* and *service* routines. Also, the function can access the same data structures as the *protect_q put* and *service* routines.
- *Put* and *service* routines cannot be called directly. They must be executed by calling STREAMS/UX utilities such as *putnext, put, putq*, or *qenable*. They cannot be called using the function pointer stored in the *q_qinfo* structure.
- STREAMS/UX applications in which multiple processes access the same stream need to know how STREAMS/UX will synchronize operations on the stream. See “Multiple Processes Accessing the Same Stream” in Chapter 3.
- Modules and drivers can allocate their own spinlocks to protect data structures. If they do, they should use the lock orders reserved for them in */usr/include/sys/semglobal.h* or */usr/conf/h/semglobal.h*: *STREAMS_USR1_LOCK_ORDER, STREAMS_USR2_LOCK_ORDER*, and *STREAMS_USR3_LOCK_ORDER*.

  The lock order is passed in the order parameter of the native HP-UX *alloc_spinlock* primitive and the hierarchy parameter of the SVR4 MP *LOCK_ALLOC* utility. The HP-UX kernel uses this information to check for deadlocks when the kernel is compiled with *SEMAPHORE_DEBUG*. When a module acquires a spinlock, the spinlock's order must be higher than the order of any spinlocks the module already holds. Modules and drivers cannot hold

spinlocks when calling some STREAMS/UX utilities. See Table 1 at the end of this chapter for more information. See the SVR4.2 Driver manual for more information about SVR4 MP hierarchies.

- To reduce contention and improve performance, you should minimize the amount of time that modules and drivers hold spinlocks.
- To improve performance, modules and drivers should verify that they are actually running on a multiprocessor system before calling the HP-UX native spinlock primitives. The SVR4 MP LOCK and UNLOCK routines described in Chapter 3 do this for the caller. If a spinlock is being used only to protect against software running on other processors, but not interrupts, modules or drivers can call the *MP_SPINLOCK* and *MP_SPINUNLOCK* macros in */usr/include/sys/spinlock.h* (or */usr/conf/h/spinlock.h*). These macros obtain only the requested spinlock if they are executing on a multiprocessor system. If a spinlock is being used to protect against both software running on other processors and interrupts, modules and drivers should check the uniprocessor flag and raise the *spl* level if they are running on a uniprocessor system. Example code is shown below.

  ```
  if (uniprocessor)
     x = splstr();
  else
     spinlock(mylock);
  ```

- Be careful when choosing a multiplexor's synchronization level. When a driver is linked under a mux, STREAMS/UX changes the driver's Stream head to be the lower mux. STREAMS/UX uses the upper mux's synchronization level for the lower mux. So if the upper mux uses global, elsewhere, or module synchronization, the lower and upper muxes can share data. If the upper mux uses queue or queue pair synchronization, the lower and upper muxes cannot share data.

  The synchronization level also influences how messages can be passed across the mux. If the upper mux uses global, elsewhere, or module synchronization, it can pass messages downward by passing the lower mux's write queue to *putq*, *put*, or *putnext*. Likewise, the lower mux can pass messages upward by passing the upper mux's read queue to *putq*, *put*, or *putnext*. If the upper mux uses queue or queue pair synchronization, it can only use *putq* and put to pass messages to the lower mux. To use *putnext*, the upper mux must ensure that the driver stays linked under the mux until after the putnext completes. Also, the lower mux can only use putq and put to pass messages to the upper mux. To use *putnext*, the lower mux must guarantee that the driver stays linked under the mux, that the mux stays open, and that modules are not pushed or popped until after the *putnext* completes.

No matter which utility is used to pass messages across the mux, you must make sure that the queues passed to the utilities are still allocated. You may also want to check that the driver is still linked under the mux.

- Follow the design guidelines in the SVR4.2 STREAMS manual. The guidelines are located at the end of these chapters: Overview of STREAMS Modules and Drivers, STREAMS Modules, STREAMS Drivers, and STREAMS Multiplexing. For STREAMS/UX, you do not need to follow some of these guidelines. However, if you ignore them, your software will not be portable to SVR4 STREAMS. For HP-UX STREAMS, you do not need to call *qprocson* or *qprocsoff* as you do for SVR4 MP STREAMS. Also, you can use synchronization levels to protect module and driver private structures instead of SVR4 MP locks and synchronization primitives. Lastly, you do not need to use SVR4 MP *canputnext* and *bcanputnext* instead of *canput* and *bcanput* on STREAMS/UX.

Porting SVR4 MP Modules and Drivers to HP-UX

Please read the previous section, “Writing MP Scalable Modules and Drivers,” before this one. If you compare the previous section to the SVR4.2 STREAMS manual, you will notice that there are some differences between SVR4 MP STREAMS and HP-UX MP STREAMS. This section discusses these differences and describes strategies for porting SVR4 MP modules and drivers to HP-UX.

Differences between SVR4 and HP-UX MP STREAMS

HP-UX STREAMS provides MP scalability differently from SVR4 MP STREAMS. There are two main differences. The first pertains to which STREAMS/UX entities run in parallel. SVR4 MP STREAMS executes put and service routines for the same queue concurrently although only one instance of a service routine can run at a time. HP-UX, unlike SVR4 MP, allows the developer to configure which STREAMS/UX entities run in parallel. The most parallelism that a STREAMS/UX developer can configure is to run entry points for different queues concurrently. Unlike SVR4 MP, HP-UX only allows one entry point for a queue to run at a time. The put and service routines for the same queue cannot run in parallel. Also, multiple instances of a queue's put or service routine cannot execute concurrently.

The second difference has to do with synchronizing access to module and driver private data structures. SVR4 MP STREAMS does not provide protection for private structures. The module or driver code uses spinlocks to synchronize access. STREAMS/UX provides protection for private structures. The developer configures the amount of concurrency for a module or driver based on the entities with which it shares data structures. For example, if all instances of a module access the same table, the programmer can configure the module so that only one instance runs at a time.

Strategies for Porting SVR4 MP Modules and Drivers to HP-UX

The best way to port SVR4 MP scalable modules and drivers to HP-UX is to change the SVR4 MP code to use the STREAMS/UX synchronization levels. First, analyze how the SVR4 MP code shares data structures, and then configure the modules and drivers to use synchronization levels which correctly serialize access to shared data. You can use defines to change module and driver spinlock calls to no-ops. This approach is likely to get the best performance, but may require much effort. Also, the STREAMS/UX synchronization levels may not be suitable for all designs.

To make porting easier, STREAMS/UX will provide support for the SVR4 MP spinlock primitives. SVR4 MP modules and drivers could be ported to HP-UX by configuring them to run with queue synchronization and leaving in the calls to SVR4 MP spinlock routines. A disadvantage of this porting strategy is that it may not achieve as much performance as the first. Some of the synchronization provided by STREAMS/UX will be redundant with the synchronization implemented by module and driver spinlocks. In some cases, a combination of these two strategies may make sense. For example, suppose several modules and drivers share the same structure, but do not access it on the main read and write paths. You can use SVR4 MP spinlocks to protect this data, but use the STREAMS/UX synchronization levels to protect other structures.

MP Synchronization Levels on a Uniprocessor

This section describes how modules and drivers can use MP synchronization levels on a uniprocessor system to protect their private data structures against interrupts. Please read “Writing MP Scalable Modules and Drivers” in this chapter and “STREAMS/UX Uniprocessor Synchronization” in Chapter 3 before reading this section.

In addition to the techniques described under “Driver and Module Synchronization” in Chapter 3, modules and drivers can use MP synchronization levels to protect their private structures against interrupts. By default STREAMS/UX configures modules and drivers to use queue pair synchronization. This is why modules and drivers do not need to raise the *spl* level to protect their data if software running on the ICS sends a message to a stream. Suppose an interrupt occurs while one of a queue pair's entry points is running. STREAMS/UX will re-schedule sending the message to the stream to after the entry point finishes executing. You can configure uniprocessor modules and drivers to use synchronization levels other than queue pair synchronization if they need more protection.

For example, you could configure a module to use module synchronization if multiple instances of the module share the same data structure, and if the module updates the structure when it is running on the ICS. If you configure the module to use module synchronization, STREAMS/UX will wait until no instances of the module are running before sending it a message. Alternatively, you could change the module to raise the spl level while accessing the shared structure.

You configure synchronization levels for modules and drivers that run on a uniprocessor system in the same way as for MP scalable modules and drivers. You must specify the synchronization level the module or driver uses, and if the module or driver requires elsewhere synchronization, you must specify a *sync* name. The *sync* name indicates which modules and drivers belong to a group. Pick a *sync* name with 8 or fewer characters, and configure the name for each member of the group. You configure the synchronization level and the sync name in either the master file $STREAMS_DVR_SYNC table or in an install function *streams_info_t* structure.

You can configure modules and drivers to use a particular synchronization level whether or not they are MP scalable, run under UP emulation, or only run on a uniprocessor system. The section “Configuring MP Scalable Modules and Drivers” in this chapter shows examples of configuring MP scalable modules and drivers to use synchronization levels. There is no difference between configuring modules and drivers which run only on a uniprocessor system and modules and drivers which run under UP emulation. Examples of configuring uniprocesor/UP emulation modules and drivers are shown below. Examples are given for both master file entries and module and driver install functions. See Chapter 5 for more information about configuring modules and drivers.

MASTER FILE $DEVICE TABLE CONFIGURATION

```
* name   handle type mask  block char
*
$DEVICE
A        Ainfo   40  2000   -1   -1   /* UP module, since 0x10000 not in mask */
B        Binfo   40  2000   -1   -1   /* UP module, since 0x10000 not in mask */
C        Cinfo   40  2000   -1   -1   /* UP module, since 0x10000 not in mask */
D        Dinfo   21  20FC   -1   116  /* UP driver, since 0x10000 not in mask */

$$$
* name   sync level         sync name
*
$STREAMS_DVR_SYNC
A        sync_module                  /* Module uses synch level */
B        sync_module                  /* Module uses synch level */
C        sync_elsewhere     netsync   /* Module uses synch level & name */
D        sync_elsewhere     netsync   /* Driver uses synch level & name */
$$$
```

INSTALL FUNCTION CONFIGURATION

```
B MODULE

static streams_info_t b_str_info = {  /* streams information */
    “B”,                              /* name */
    -1,                               /* major number */
    { &brinit, &bwinit, NULL, NULL },  /* streamtab */
    STR_IS_MODULE | STR_SYSV4_OPEN,    /* *****NOTE***** MGR_IS_MP not specified */

    SQLVL_MODULE,                     /* *****NOTE***** synch level specified */
     “”,                              /* sync name */
}

int
B_install()
{
   int retval;

   return(str_install(&b_str_info));

}


D DRIVER

static drv_info_t d_drv_info = {   /* driver information */
     “D”,                          /* name */
     “pseudo”,                     /* class */
     DRV_CHAR | DRV_PSEUDO,        /* *****NOTE***** DRV_MP_SAFE flag not specified */

     -1,                           /* block major number */
     -1                            /* dynamically assigned character major number */

    NULL, NULL, NULL,           /* cdio, gio_private,and cdio_private structures
*/
                                        structures */
}


static drv_ops_t d_drv_ops = {      /* driver entry points */
      NULL,                         /* open */
      NULL,                         /* close */
      NULL,                         /* strategy */
      NULL,                         /* dump */
      NULL,                         /* psize */
      NULL,                         /* mount */
      NULL,                         /* read */
      NULL,                         /* write */
      NULL,                         /* ioctl */
      NULL,                         /* select */
      NULL,                         /* option1 */
      NULL, NULL, NULL, NULL,       /* reserved entry points */
      0,                            /* device flags */
}
```

```
static streams_info_t d_str_info =  {  /* streams information */
      “D”,                              /* name */
        -1,                             /* dynamically assigned major number */
      { &drinit, &dwinit, NULL, NULL}, /* streamtab */
      STR_IS_DEVICE | STR_SYSV4_OPEN, /* *****NOTE***** MGR_IS_MP flag not specified */
      SQLVL_ELSEWHERE,                  /* *****NOTE***** synch level specified*/
      “netsync”,                        /* *****NOTE***** sync name specified */
}

int
D_install()
{
    int retval;

    /* Configure driver and obtain dynamically assigned major number. */
    if ((retval = install_driver(&d_drv_info, &d_drv_ops)) != 0)
       return(retval);


    /* Configure streams specific parameters. */
    if ((retval = str_install(&d_str_info)) != 0) {
       uninstall_driver(&d_drv_info);
       return(retval);
    }

    /* Success */
    return 0;

}
```

The following table indicates if spinlocks can be held across calls to different STREAMS/UX utilities. Also, it specifies if the SVR4 MP STREAMS/UX utilities have the same restrictions.

Table 1 Holding Module or Driver Defined Spinlocks While Calling Utilities

Utility	Spinlocks Can Be Held Across Call?	Differs From SVR4 MP?
adjmsg	Yes	No
allocb	Yes, if use STREAMS/UX user lock orders.	No
backq	Yes	No
bcanput	Yes, if use STREAMS/UX user lock orders.	No
bcanputnext	Yes, if use STREAMS/UX user lock orders.	No
bcopy	Yes	No
bufcall	Yes, if use STREAMS/UX user lock orders.	
bzero	Yes	No
canput	Yes, if use STREAMS/UX user lock orders.	No
canputnext	Yes, if use STREAMS/UX user lock orders.	No
cmn_err	No	Yes
copyb	Yes, if use STREAMS/UX user lock orders.	No
copymsg	Yes, if use STREAMS/UX user lock orders.	No
datamsg	Yes	No
delay	No	No
drv_getparm	Yes	No
drv_priv	Yes	No
dupb	Yes, if use STREAMS/UX user lock orders.	No
dupmsg	Yes, if use STREAMS/UX user lock orders.	No
enableok	Yes, if use STREAMS/UX user lock orders.	No

Table 1 Holding Module or Driver Defined Spinlocks While Calling Utilities

Utility	Spinlocks Can Be Held Across Call?	Differs From SVR4 MP?
esballoc	Yes, if use STREAMS/UX user lock orders.	No
esbbcall	Yes, if use STREAMS/UX user lock orders.	No
flushband	Yes, if use STREAMS/UX user lock orders (flushband may call user *esballoc* free routines).	No
flushq	Yes, if use STREAMS/UX user lock orders (flushq may call user *esballoc* free routines).	No
freeb	Yes, if use STREAMS/UX user lock orders (freeb may call user *esballoc* free routines).	No
freemsg	Yes, if use STREAMS/UX user lock orders (*freemsg* may call user *esballoc* free routines).	No
freezestr	Yes	No
getadmin	Yes, if use STREAMS/UX user lock orders.	No
getmid	Yes, if use STREAMS/UX user lock orders.	No
getmajor	Yes	No
getminor	Yes	No
getq	Yes, if use STREAMS/UX user lock orders.	No
insq	Yes, if use STREAMS/UX user lock orders.	No
itimeout	Yes, if use STREAMS/UX user lock orders.	No
kmem_alloc	Yes, if use STREAMS/UX user lock orders and KM_NOSLEEP.	No
kmem_free	Yes, if use STREAMS/UX user lock orders.	No
linkb	Yes	No
LOCK	Yes, if use lock orders correctly.	No

Table 1 Holding Module or Driver Defined Spinlocks While Calling Utilities

Utility	Spinlocks Can Be Held Across Call?	Differs From SVR4 MP?
LOCK_ALLOC	No	Yes
LOCK_DEALLOC	Yes, if use STREAMS/UX user lock orders.	No
major	Yes	No
makedev	Yes	No
makedevice	Yes	No
max	Yes	No
min	Yes	No
minor	Yes	No
msgdsize	Yes	No
msgpullup	Yes, if use STREAMS/UX user lock orders.	No
noenable	Yes, if use STREAMS/UX user lock orders.	No
OTHERQ	Yes	No
pcmsg	Yes	No
pullupmsg	Yes, if use STREAMS/UX user lock orders.	No
put	No	No
putbq	Yes, if use STREAMS/UX user lock orders.	No
putctl	No	No
putctl1	No	No
putctl2	No	No
putnext	No	No
putnextctl	No	No
putnextctl1	No	No

Table 1 Holding Module or Driver Defined Spinlocks While Calling Utilities

Utility	Spinlocks Can Be Held Across Call?	Differs From SVR4 MP?
putnextctl2	No	No
putq	Yes, if use STREAMS/UX user lock orders, and does not pass driver's read queue or lower mux's write queue.	Yes
qenable	Yes, if use STREAMS/UX user lock orders.	No
qprocsoff	Yes	Yes
qprocson	Yes	Yes
qreply	No	No
qsize	Yes	No
RD	Yes, if use STREAMS/UX user lock orders.	No
rmvb	Yes	No
rmvq	Yes, if use STREAMS/UX user lock orders.	No
SAMESTR	Yes, if use STREAMS/UX user lock orders.	No
sleep	No	No
spln	No	Yes
splstr	No	Yes
streams_put	No	No
streams_get_sleep_lock	Yes, if use STREAMS/UX user lock orders.	No
strlog	No	Yes
strqget	Yes, if use STREAMS/UX user lock orders.	No
strqset	Yes, if use STREAMS/UX user lock orders.	No
SV_ALLOC	Yes, if use STREAMS/UX user lock orders and KM_NOSLEEP.	No

Table 1 Holding Module or Driver Defined Spinlocks While Calling Utilities

Utility	Spinlocks Can Be Held Across Call?	Differs From SVR4 MP?
SV_BROADCAST	Yes, if use STREAMS/UX user lock orders.	No
SV_DEALLOC	Yes, if use STREAMS/UX user lock orders.	No
SV_WAIT	No, except for *lkp* parameter lock.	No
SV_WAIT_SIG	No, except for *lkp* parameter lock.	No
testb	Yes, if use STREAMS/UX user lock orders.	No
timeout	Yes, if use STREAMS/UX user lock orders.	No
TRYLOCK	Yes	No
unbufcall	No	No
unfreezestr	Yes	No
unlinkb	Yes	No
UNLOCK	Yes	No
untimeout	Yes, if locks can be held across call to timeout callback function.	No
unweldq	Yes, if use STREAMS/UX user lock orders.	No
vtop	Yes	No
wakeup	Yes, if use STREAMS/UX user lock orders.	No
weldq	Yes, if use STREAMS/UX user lock orders.	No
WR	Yes, if use STREAMS/UX user lock orders.	No

5

How to Compile and Link STREAMS/UX Drivers, Modules, and Applications

This chapter describes how STREAMS/UX drivers and modules can be added to the HP-UX kernel, and how STREAMS/UX TLI and XTI applications can be compiled and linked.

Compiling STREAMS/UX Drivers and Modules

The steps for compiling STREAMS/UX drivers and modules follow.

1 Include the appropriate STREAMS/UX include files in the driver and module sources. Table 2 describes the files. Drivers and modules are compiled in the */usr/conf* directory. They contain include statements with relative path names. The table shows the path names.

Table 2 **STREAMS/UX and TPI Include Files**

Include File	Use
"../h/stream.h"	Needed by all drivers and modules.
"../h/stropts.h"	Needed by all drivers and modules.
"../h/strlog.h"	Needed by drivers and modules that call *strlog*. Note that *log.h* and *syslog.h* are not needed. STREAMS/UX does not support priority and facility codes.
"../h/strstat.h"	Needed by drivers and modules that use the *qi_mstat* field of the *qinit* structure to maintain statistics.
"../h/strenv.h"	Needed by drivers and modules that use DKI functions.
"../h/cmn_err.h"	Needed by drivers and modules that use cmn_err().
"../h/tihdr.h"	Needed by drivers and modules that use TPI.

2 If you are only adding modules, you will need to archive those modules into a library.

3 Compile the sources in */usr/conf* with the appropriate options. Create a directory under */etc/conf* and place your source files in this directory. Use the following command line with appropriate substitutions to compile your source code.

```
@${CC}  -I.  -c  ${CFLAGS}  ${NOGLOOPTS}  $(your_file).c
```

Compile each of your modules and archive the object files into a library using the *ar* command. It is best to place all of your driver and module code into the same library. In the example below, *libexample1.a* is the name of the library and *obj*.o* are the object files:

```
rm -f libexample1.a
ar -r libexample1.a  ojb1.o  obj2.o  ... objn.o
```

Linking STREAMS/UX Drivers and Modules into the Kernel

Linking STREAMS/UX drivers and modules into the kernel is a multi-step process. A summary of the steps is:

1. Create or modify your master file to reflect changes.
2. Add a driver header with the information previously located in the */etc/master* file into the *etc/master.d* directory.
3. Add a driver install routine for both STREAMS drivers and STREAMS modules ("driver" in the case of the STREAMS subsystem refers to both STREAMS drivers and STREAMS modules).
4. Adjust any STREAMS/UX tunables if necessary.
5. Create your library and copy it to */usr/conf/lib*.
6. Re-generate your kernel using mk_kernel(1).
7. Once the system is re-booted, use *lsdev(1M)* to determine the value of any dynamically-assigned major numbers, if applicable.
8. Create device files with *mknod(1M)*.

Details about the Driver Header, Driver Install Routine, and lsdev(1) follow.

Adding Driver Header and Driver Install Routine

The STREAMS driver writer must add a driver header and a driver install routine for their STREAMS drivers and modules. The driver header consists of three data structure declarations (for a STREAMS driver and actually only one for a STREAMS module). The driver install function will get called by the I/O system to "install" your pseudo driver into the I/O subsystem tables. The driver header essentially contains the information previously contained in the master file.

The main job of your driver install routine is to call one or both of the functions, *install_driver* (CDIO3) and/or *str_install()*.

For a STREAMS driver, your driver install routine will need to call both the function *install_driver* (CDIO3) and *str_install()*. And for a STREAMS module, your driver install routine will only need to call the *str_install()* routine.

The call to *install_driver()* initializes the *cdevsw* entry points and d_flags for your STREAMS driver. The call to the *str_install()* function fills out either the *dmodsw* (for a STREAMS driver) or the *fmodsw* (for a STREAMS module) switch tables used by the STREAMS subsystem.

NOTE:

The *str_install()* function will replace the *open, close, read, write, ioctl, select,* and *option1 cdevsw* entry points with the STREAMS/UX-specific entry points. So it is best to use NULLs in the *drv_ops_t* structure as illustrated in the example later in this section.

Keep in mind that you can call your *driver_link* routine from the driver install to perform any necessary driver initialization tasks. You should not perform any operations which require returning error conditions or data. Plus, it is best to keep driver install routines small and clean to avoid bootup problems.

If you are writing MP STREAMS drivers and STREAMS modules, refer to Chapter 4 for specific MP requirements. Chapter 4 provides examples of driver headers and driver install routines relating to MP drivers and modules.

The driver header can be declared in either a *.h* or in the *.c* file that contains the driver install entry point. The driver install entry point MUST be in a *.c* file.

For both STREAMS drivers and STREAMS modules, the following include files contain the needed structures and defines:

```
#include "../h/conf.h"
#include "../h/stream.h"
```

Streams Driver

For STREAMS drivers, the following data structures will need to be declared in the *.h* or *.c* file: *drv_info_t, drv_ops_t* and *streams_info_t*.

An example of these declarations using the STREAMS test driver, “tlo” is as follows: (The STREAMS *tlo* test driver is used only as an example throughout this section. Please tailor this example to your specific driver configuration).

drv_info_t

```
static drv_info_t tlo_drv_info = {
    “tlo”,                                  /* driver name */
    “pseudo”,                               /* driver class */
    DRV_CHAR | DRV_PSEUDO | DRV_MP_SAFE,    /* flages */
    -1,                                     /* block major number */
    -1,                                     /* char major number */
    NULL, NULL, NULL,                        /* cdio, gio_private and
cdio_
                                                private  always  NULL
*/};
```

drv_ops_t

```
static drv_ops_t tlo_drv_ops = {
    NULL,                       /* d_open     */
    NULL,                       /* d_close    */
    NULL,                       /* d_strategy */
    NULL,                       /* d_dump     */
    NULL,                       /* d_psize    */
    NULL,                       /* d_mount    */
    NULL,                       /* d_read     */
    NULL,                       /* d_write    */
    NULL,                       /* d_ioctl    */
    NULL,                       /* d_select   */
    NULL,                       /* d_option1  */
    NULL, NULL, NULL, NULL,     /* reserved entry points */
    NULL,                       /* d_flags */
};
```

streams_info_t

```
static streams_info_t tlo_str_info = {
    “tlo”,                          /* name */
    -1,                             /* dynamic major number */
    { &tlorinit, &tlowinit, NULL
        NULL },                     /* streamtab */
    STR_IS_DEVICE | MGR_IS_MP |     /* streams flags */
    STR_SYSV4_OPEN,
    SQLVL_QUEUE,                    /* sync level */
    “”,                             /* elsewhere sync name */
};
```

The definitions of the *streams_flags* used in the *streams_info_t* structure are (see stream.h and conf.h):

```
STR_IS_DEVICE   /* Indicates a driver is being installed */
STR_IS_MODULE   /* Indicates a module is being installed */
STR_SYSV4_OPEN  /* Indicates SVR4 open is being used, SVR3 open
                    is default */
MGR_IS_MP       /* Module/driver is MP-scalable */
```

The *sync level* used in the *streams_info_t* structure is one of the following defined in stream.h:

SQLVL_DEFAULT
SQLVL_GLOBAL
SQLVL_ELSEWHERE
SQLVL_MODULE
SQLVL_QUEUEPAIR
SQLVL_QUEUE

For STREAMS drivers, a driver install routine needs to be added to the .c file for your driver. This function MUST be called *xxxx_install*, where *xxxx* is the driver handle used for your driver. Exactness is needed so that your driver install routine is correctly called by the I/O subsystem during bootup.

Illustrated below is an example of the driver install routine for the example tlo driver.

```
int
tlo_install()
{
    int retval;

    if ((retval = install_driver (&tlo_drv_info, &tlo_drv_ops)) !=0)
            return (retval);

    if ((retval = str_install (&tlo_str_info)) !=0) {
            uninstall_driver (&tlo_drv_info);
            return (retval);
    }
    return (0); /* return success */
}
```

In this tlo example, a major number of -1 was defined in both the *tlo_drv_info* and *tlo_str_info* structure declarations. This invokes the dynamic major facility. When using this facility, you will need to obtain the system

assigned “dynamic” major number by running the *lsdev(1)* command after the system has rebooted with the kernel that includes your driver. There are details later in this section on *lsdev(1)*.

STREAMS Module

For STREAMS modules, steps identical to those executed for a STREAMS driver are needed, but with the following exceptions:

For the driver header, you only need to declare a *streams_info_t* structure. This is because STREAMS modules do not have any cdevsw-related information. They only have STREAMS-specific information and this is configured by calling *str_install()* with a defined *streams_info_t*.

For the driver install routine, you need only to call the *str_install()* function. There is no need to call *install_driver*(CDIO3).

An example of these declarations using the STREAMS test module, “lmodb,” is as follows: (The STREAMS *lmodb* test module is used only as an example. Please tailor this example to your specific module configuration).

streams_info_t

```
static streams_info_t lmodb_str_info = {
  “lmodb”,                                  /* name */
  -1,                                       /* major number */
  { &lmodbrinit, &lmodbwinit, NULL, NULL},/* streamtab */
  STR_IS_MODULE,                            /* streams flags */
  SQLVL_QUEUEPAIR,                          /* sync level */
  “”,                                       /* elsewhere sync name */
};
```

The *streams_flags* and the *sync level* to be used in the *streams_info_t* structure are the same as illustrated above in the “STREAMS Driver” section, except we are using “STR_IS_MODULE,” instead of “STR_IS_DEVICE.”

Illustrated below is an example of the driver install routine required for a STREAMS module, using the example *lmodb* module.

```
int
lmodb_install()
{
    int retval;

    if ((retval = str_install (&lmodb_str_info)) != 0)
    {
         return (retval);
    }
    return 0; /* return success */
};
```

Modifying Your Master File

In 10.0, the */etc/master* file is replaced by a collection of files located in */usr/conf/master.d* directory. It is recommended that you create your own individual master file, calling it something appropriate. See */usr/conf/master.d/streams* for the master file used by the STREAMS/UX framework. You may use the STREAMS master file as a template for creating your specific master file.

You will need to add entries for each of your STREAMS drivers to the $DRIVER_INSTALL section of your master file. See the master(4) manpage for a description of the master file section layouts and dynamic major numbers.

An example $DRIVER_INSTALL section from the STREAMS/UX master file is as follows:

```
$DRIVER_INSTALL
************************************************************************
* Driver install table
*
* This table contains the name of drivers which have converged I/O header
* structures and install entry points. Drivers in this table should not
* be defined in the driver table above.
************************************************************************
* Driver    Block major     Char major
clone           -1             72
strlog          -1             73
sad             -1             74
echo            -1            116

* Example driver entry which must use dynamic major numbers indicated by -1
tlo             -1             -1
```

In addition, you will also need to add additional entries for any STREAMS modules to the $DRIVER_INSTALL section as well. Using the example *lmodb* module:

```
$DRIVER_INSTALL
************************************************************************
* Driver install table
*
* This table contains the name of drivers which have converged I/O header
* structures and install entry points. Drivers in this table should not
* be defined in the driver table above.
************************************************************************
* Driver     Block major    Char major
lmodb           -1             -1
```

When adding an entry to the $DRIVER_INSTALL section of your master file, do NOT add an entry to the $DEVICE section of your master file. This will result in a possible conflict (such as duplicate major numbers) and/or a lack of a call to your driver install routine at bootup. The only way to use the dynamic major number facility is to configure your STREAMS driver as documented in this section.

For more details on driver headers and driver install routines, please read the *HP-UX Driver Development Guide* (P/N 98577-90000-E1).

Dynamically-Assigned Major Numbers and lsdev(1)

When using the dynamic major number facility, you will need to determine which major number was assigned to your driver during bootup, by consulting *lsdev(1)*. Once the system is booted with your new kernel, run the *lsdev(1)* command. See the lsdev(1) manpage for all the option details, but in brief you can use *lsdev(1)* as shown below.

NOTE: For STREAMS-clonable devices, use 72 for the major and your driver's assigned major number for the minor number.

```
lsdev -h -d <your_driver_name_here>
```

(the -h means that *lsdev* does not print a header)

and use the result for your mknod(1M):

```
mknod /dev/<device_file_name> c 72  0x<dyn_major result>

mknod /dev/<device_file_name> c <dyn_major result> 0x0
```

The first *mknod* command is for a clonable device. The second is for a non-clonable device.

Compiling and Linking STREAMS/UX Applications

Follow these steps for compiling and linking STREAMS/UX applications:

1 Include the appropriate header files. The following header files may be found in */usr/include* or */usr/include/sys*. Those found in */usr/include* are pointers to the files found in */usr/include/sys*. POSIX compliance required the files to be moved to the sys directory so pointer files were established for source backward compatibility.

Table 3 STREAMS/UX Include Files

Include File	Use
<stropts.h> or <sys/stropts.h>	Needed by all STREAMS/UX applications.
<poll.h> or <sys/poll.h>	Needed by programs that use *poll*.
<sad.h> or <sys/sad.h>	Needed by programs that open the *sad* driver.
<strlog.h> or <sys/strlog.h>	Needed by programs that open the *strlog* driver. Note that *log.h* and *syslog.h* are not needed. STREAMS/UX does not support priority and facility codes.

2 Compile the source files. There are no required compiler or linker options for STREAMS/UX. See the appropriate compiler man page for which options to choose.

NOTE: The STREAMS/UX system calls have been made thread-safe and are part of *libc*. If you link the application with the threads library, *libcma*, then you may make use of the threads utilities. No special considerations are needed for STREAMS-based applications, though it is recommended that the developer have a thorough understanding of threads principles before coding such an application using the STREAMS/UX system calls. Please read the following section for additional caveats for coding threaded applications.

Compiling and Linking TLI/XTI Applications and Threads

As with the STREAMS/UX system calls, compiling and linking a TLI or XTI application requires no special compile or linking options. Choose the appropriate include files from the table below and compile. Link your application with either the TLI library, *libnsl_s.a* or *libnsl_s.sl*, or the XTI library, *libxti.a* or *libxti.sl*. Both libraries are in */usr/lib*.

Table 4 TLI/XTI Include Files

Include File	Use
<xti.h> or <sys/xti.h>	Needed by all XTI applications.
<tiuser.h> or <sys/tiuser.h>	Needed by all TLI applications.
<poll.h> or <sys/poll.h>	Needed by programs that use poll.
<stropts.h> or <sys/stropts.h>	Needed by programs that use the STREAMS/UX interface to perform operations such as pushing modules onto a stream.

These libraries have been made thread-safe, that is, these libraries may be used with both non-threaded and multi-threaded applications. Please see OSF/DCE documentation for the POSIX threads library calls that may be used.

The following caveats apply to this release of these libraries:

- When a thread is executing within a TLI/XTI library call, the thread may not be canceled. The library will turn both general and asynchronous cancellation off during execution. This is necessary to avoid corruption of internal mutex structures.
- The global variable *t_errno* and the function *t_strerror()* will return values on a per-thread basis. These values are stored in thread-specific pointers via the *pthread_setspecific()* and *pthread_getspecific()* functions.
- It is possible to deadlock a process should the application attempt to execute in loopback using two threads within the same process' address space. It is

recommended that for loopback applications, the sending and receiving threads be in separate processes which will avoid any deadlock situation.

NOTE: The libraries use two levels of internal locks and it is only during the small time frame between obtaining and releasing the locks that a deadlock can occur.

- The *libcma.a* or *libcma.sl* library must be linked into the application before either the *libnsl_s.a* or the *libxti.a* libraries are linked.
- The include file, *pthread.h*, must be the first include file defined within an application to have all entry points properly mapped.
- Independent of the TLI/XTI libraries, if you cancel a thread and it was either waiting on a mutex or a condition variable, it is best to consider either the mutex or the condition variable as corrupted and either re-initialize or destroy and recreate them.

Here are some basic tips on coding TLI/XTI multi-threaded applications.

- The *pthreads* library is a user space library. Threads execute using either the default round-robin scheduling mechanism or a scheduling mechanism that the application controls. The default order of execution is not predictable nor should it be relied upon. For instance, if a thread spawns multiple threads, the new threads will not be allowed to execute until either the initiating thread executes a blocking call, executes a *pthread_yield*(), or its time slice expires. It is recommended that a thread executing a *pthread_create*() issue a *pthread_yield*() and possibly a *pthread_join*() to allow the other threads a chance to execute and finish their tasks before continuing its processing.

  For TLI/XTI, this technique is useful for a responder application which listens for incoming connections and creates a new thread to complete the connection. In this case, the responder could either yield to the new thread or could continue to listen for incoming connect indications until there are no pending indications. At this time it yields or executes a poll call that will block, which allows the other threads to be scheduled for execution. This avoids the potential TLOOK error condition should another indication arrive before the previous one is processed and cleared.

  Another example is if the responder detects a POLLHUP condition exists and creates a thread to handle the disconnect, and then continues to execute. The result could be poll() detecting this condition occurring multiple times when it really only exists once. It is recommended that the thread be coded such that either a *pthread_yield*() is immediately executed following the *pthread_create*() or if the responder thread is handling multiple connections, it executes a *pthread_join*() and waits for the disconnect thread's completion.

- If using condition variables, a *pthread_cond_signal*() must be sent for each thread waiting on that condition. Condition variables are useful for synchronizing activity on a single endpoint where multiple threads may be attempting to manipulate that endpoint. Another use is coordinating multiple endpoints that need to arrive at a particular state before proceeding. This contrasts with mutex usage which is better suited for critical data or code section protection.

  An example would be if an application were replicating data at multiple sites, each thread would drive its appropriate endpoint to the state before the final commit is ready. When the controlling thread has detected that all endpoints are currently waiting at the same condition variable, the *pthread_cond_signal*() could be sent to each thread with the controlling thread waiting until the threads are complete before releasing the shared memory buffer.

- If the application is utilizing the *poll()* system call, the application will need to have error handling code in each thread to avoid unnecessary processing. For example, if multiple threads are sending data to a single endpoint and that endpoint becomes flow-controlled, when the flow-control condition is relieved, the *poll()* system call will return that the endpoint is writable. At this point one or more threads could be scheduled to execute which may result in one thread succeeding with the rest returning TFLOW errors.

- If a thread is exiting, it is recommended that the thread call *pthread_detach*() is used to release any memory that has been allocated for that thread's usage. If the detach is not performed, that memory can be lost and the application could experience memory shortage problems. Once the process is terminated, all memory should be returned to the system.

6

Debugging STREAMS/UX Modules and Drivers

Introduction

This chapter describes tools for debugging STREAMS/UX modules and drivers. STREAMS/UX supports many System V tools, and provides new ones. This chapter contains the following:

- An overview of the System V tools supported by HP-UX.
- A description of a new tool, *strdb*, that displays STREAMS/UX data structures in running systems and HP-UX core dumps. Examples are included to show the use of *strdb* in debugging driver and application problems.
- An in depth discussion of an HP-UX tool, *adb*, that helps programmers analyze core dumps. Examples show how to use *adb* and *strdb* to debug STREAMS/UX drivers and modules.

Other sections of this manual also contain debugging information. You should run the *strvf* verification tool to check that STREAMS/UX is properly installed before trying to debug modules and drivers. The STREAMS/UX Synchronization section of Chapter 3 contains module and driver programming guidelines. Read through these guidelines and the design guidelines in Chapter 7 of the *UNIX System V Release 4 Programmer's Guide: STREAMS* before testing modules and drivers.

System V Debugging Tools Supported by STREAMS/UX

STREAMS/UX supports many of the System V STREAMS debugging tools. Refer to Appendix D in the SVR4PG manual for a description of the System V tools.

STREAMS/UX Tracing and Logging

STREAMS/UX supports tracing and logging. See Appendix D and the strace(1M), strclean(1M), strerr(1M), and log(7) man pages in the SVR4PG manual for more information about these tools. Some differences exist in the user interfaces of these tools on HP-UX. These differences are described in Chapter 3 of this manual and the corresponding HP-UX man pages.

cmn_err() and printf()

HP-UX supports the DKI function *cmn_err()*. See Appendix D of the SVR4PG manual and the *Unix System V Release 4 Device Driver Interface/Driver-Kernel Interface (DDI/DKI) Reference Manual* for more information about *cmn_err()*.

Also, HP-UX supports *printf* for STREAMS/UX modules and drivers. If a STREAMS/UX module or driver calls *printf*, HP-UX prints the requested message on the system console, and stores the message in the *dmesg* buffer.

Dump Module Example

The SVR4PG manual presents a STREAMS dump module in Appendix D. The dump module traces messages flowing into and out of another STREAMS module. Appendix D contains the module source code. Programmers can copy and tailor the code to develop their own debugging tool for HP-UX. The sample master file entry and dump module include statements must be changed for HP-UX. See Chapter 2 and Chapter 5 of this manual for more information about the HP-UX master file and STREAMS/UX include statements.

strdb and adb

STREAMS/UX provides *strdb* for debugging. *strdb* can be used with the HP-UX *crash* and *adb* tools for debugging.

STREAMS/UX Debugging Tool

HP-UX provides the *strdb* tool for examining STREAMS/UX data structures in the kernel. *strdb* is an interactive tool. You run the *strdb* program, and then enter commands to see data structures. This section describes *strdb* commands and shows examples of using *strdb* to find STREAMS/UX driver problems. The *strdb* man page summarizes *strdb* commands.

Running strdb

The syntax for the *strdb* command is:

```
strdb [vmunix_executable_file_name][vmunix_core_file_name]]
```

STREAMS/UX programmers can run *strdb* to look at snapshots of STREAMS/UX data structures in the kernel while HP-UX is running. Also, programmers can run *strdb* to look at STREAMS/UX data structures in a vmunix core file. To see STREAMS/UX data structures while the system is running, enter:

```
strdb
```

Sometimes the system is booted using a different kernel than */stand/vmunix*, for example */vmunix.prev*. In this case, run *strdb* by entering:

```
strdb /vmunix.prev
```

To look at STREAMS/UX data structures in a core file, pass the name of the hp-ux program and core files to *strdb*. For example, if the program and core files have the paths */var/adm/vmunix.0* and */var/adm/vmcore.0*, enter:

```
strdb /var/adm/vmunix.0 /var/adm/vmcore.0
```

strdb Commands

After invoking *strdb*, you can enter commands to look at STREAMS/UX data structures. *strdb* runs in two modes, primary and STREAMS/UX subsystem. Each mode provides different commands.

Primary mode commands change the characteristics of the *strdb* session. For example, one command turns on logging to a file. Primary mode commands also allow you to navigate through STREAMS/UX data structures. When *strdb* starts up, you are in primary mode. You switch to STREAMS/UX subsystem mode by entering the *:S* command.

STREAMS/UX subsystem mode commands report what STREAMS/UX are configured and active on the system. Also, the *qh* command allows you to begin examining a particular stream's queues. This command displays a selected stream head read queue. In addition, it puts you into primary mode so that you can use the primary mode navigation commands to traverse the rest of that stream's queues. All the commands for both modes are listed later in this chapter.

In a typical *strdb* session, you might do the following:

1. Start *strdb* (you are in primary mode).
2. Execute the *:S* command to enter STREAMS/UX subsystem mode.
3. Use STREAMS/UX subsystem mode commands to find the active stream you want to examine.
4. Execute the *qh* command to display the selected stream head read queue. This puts you in primary mode.
5. Enter primary mode navigation keys to display fields in the stream head read queue, and traverse the rest of that stream's queues.

STREAMS/UX Subsystem Commands

When you first enter *strdb*, *strdb* prints a message saying that you have not yet specified a stream to display. You can enter the *:S* command to get into the STREAMS/UX subsystem mode. *strdb* will display the following help menu.

```
STREAMS subsystem help commands
?                  - show this help menu
h                  - show this help menu
la  'name'          - list all active STREAMS on device 'name'
ll  'name' 'minor' - list all drivers linked under the STREAMS
                      driver 'name' and minor number 'minor'
lm  'name' 'minor' - list all modules pushed on STREAMS device
                     'name'and whose minor number is 'minor'
lp  'name' 'minor' - list all drivers persistently linked under
                      the STREAMS device 'name' and minor number
                     'minor'
```

```
q                 - quit the STREAMS subsystem commands
qc 'driver' 'file' - print 'driver' read / write side qcount to
                     'file'
qh  'name' 'minor' - display STREAM head queue structure
                     for device 'name' and minor number 'minor'
s   [m | d]        - Option d lists all the STREAMS drivers
                     configured in the system. Option m lists
                     all the modules configured in the system
v                 - print   version   of   STREAMS   structures
displayed
```

The *?*, *h*, *q*, *v*, *s*, *la*, *lm*, *ll*, *lp*, *qc*, and *qh* commands are available in subsystem mode. To execute these commands, enter the command at the ":" prompt. The commands help you find the stream that you want to examine. The commands are described below.

? and h Commands

Enter the *?* or *h* command to see the help menu for STREAMS/UX subsystem mode. *strdb* prints the text shown below.

```
?

STREAMS subsystem help commands
?                 - show this help menu
h                 - show this help menu
la  'name'         - list all active STREAMS on device 'name'
ll  'name' 'minor'- list all drivers linked under the STREAMS
                     driver 'name' and minor number 'minor'
lm  'name' 'minor' - list all modules pushed on STREAMS device
'name'
                     and whose minor number is 'minor'
lp  'name' 'minor'- list all drivers persistently linked under
                     the STREAMS device 'name' and minor number
                     'minor'
q                 - quit the STREAMS subsystem commands
qc 'driver' 'file'- print 'driver' read / write side qcount to
file
qh  'name' 'minor'- display STREAM head queue structure for
                     device 'name' and minor number 'minor'
s   [m | d]       - Option d lists all the STREAMS drivers
                     configured in the system. Option m lists
                     all the modules configured in the system
v                 - print   version   of   STREAMS   structures
displayed
```

q Command

Enter the *q* command to exit STREAMS/UX subsystem mode and enter primary mode. This is shown below.

```
q

                    No current structure                    S:0
```

v Command

Enter the *v* command to display the version of STREAMS/UX data structures. This version should always be Release V 4.0. An example is shown below.

```
v

STREAMS Version based on Release V 4.0
```

s Command

Enter the *s [m|d]* command to see the STREAMS/UX modules and drivers configured into the system. These are the modules and drivers included in the multiuser *S800* file or the workstation *dfile*. Specify either *m* to see the modules or *d* to see the drivers. Examples are shown below.

```
s m

List of MODULES

timod
tirdwr
lmodb
lmode
lmodt
lmodr
lmodc
sc
bufcall


s d

List of DRIVERS

      clone  MAJOR =  72
     strlog  MAJOR =  73
        sad  MAJOR =  74
         lo  MAJOR =  75
        tmx  MAJOR =  77
       tidg  MAJOR =  78
       tivc  MAJOR =  79
```

```
       loop  MAJOR = 114
         sp  MAJOR = 115
   test_wel  MAJOR = 130
```

la Command

Enter the *la* command to see a list of opened streams for a driver. Also, enter the name of the driver. This name can be obtained from *s* command output. An example is shown below.

```
la tivc

tivc MAJOR = 79
ACTIVE Minor 0x00002f  Stream head RQ = 0x676a00
ACTIVE Minor 0x00000f  Stream head RQ = 0x663300
ACTIVE Minor 0x00000e  Stream head RQ = 0x6a5900
ACTIVE Minor 0x00002e  Stream head RQ = 0x71f800
ACTIVE Minor 0x00004e  Stream head RQ = 0x6ccf00
ACTIVE Minor 0x00000d  Stream head RQ = 0x67b300
ACTIVE Minor 0x00004d  Stream head RQ = 0x73c700
ACTIVE Minor 0x00002d  Stream head RQ = 0x728800
ACTIVE Minor 0x00004c  Stream head RQ = 0x74f600
ACTIVE Minor 0x00000c  Stream head RQ = 0x68d100
ACTIVE Minor 0x00002b  Stream head RQ = 0x730a00
```

lm Command

Enter the *lm* command to see a list of the modules pushed onto a driver. You must specify the driver name and the minor number. The minor number can be obtained from the *la* command output. An example is shown below.

```
lm tivc 47

STREAM Head
timod
Driver tivc
```

ll Command

Enter the *ll* command to see a list of drivers linked under a multiplexor. You must enter the multiplexor name and the minor number. The multiplexor name can be obtained from the *s* output. The minor number is from the *la* output. An example is shown below.

```
ll tmx 0

lo MAJOR =  75 minor = 2
lo MAJOR =  75 minor = 1
lo MAJOR =  75 minor = 0
```

lp Command

Enter the *lp* command to see a list of drivers persistently linked under a multiplexor. You must enter the multiplexor name and the minor number. An example is shown below.

```
lp tmx 1

lo MAJOR =  75 minor = 2
lo MAJOR =  75 minor = 1
lo MAJOR =  75 minor = 0
```

qc Command

Enter the *qc* command to display the *q_count* field of a driver's read and write queues. The *q_count* field contains the number of bytes of data in the messages on the queue. The command will show the *q_count* values for all the opened streams of the requested driver. You must enter the driver name and the name of a file to contain the *q_count* values. *strdb* will create the specified file and write the *q_count* values into it. An example is shown below.

```
qc tmx stat
```

<< exit from strdb >>

```
%more stat
MINOR = 5
WQ = 0x40026760, WQ_count = 0, RQ = 0x40026760, RQ_count = 4214
WQ = 0x40026760, WQ_count = 0, RQ = 0x40026760, RQ_count = 0
MINOR = 4
WQ = 0x40026760, WQ_count = 0, RQ = 0x40026760, RQ_count = 842
WQ = 0x40026760, WQ_count = 0, RQ = 0x40026760, RQ_count = 0
MINOR = 1
WQ = 0x40026760, WQ_count = 0, RQ = 0x40026760, RQ_count = 930
WQ = 0x40026760, WQ_count = 0, RQ = 0x40026760, RQ_count = 0
MINOR = 0
WQ = 0x40026760, WQ_count = 0, RQ = 0x40026760, RQ_count = 0
WQ = 0x40026760, WQ_count = 0, RQ = 0x40026760, RQ_count = 0
MINOR = 3
WQ = 0x40026760, WQ_count = 0, RQ = 0x40026760, RQ_count = 3970
WQ = 0x40026760, WQ_count = 0, RQ = 0x40026760, RQ_count = 0
MINOR = 2
WQ = 0x40026760, WQ_count = 0, RQ = 0x40026760, RQ_count = 1300
WQ = 0x40026760, WQ_count = 0, RQ = 0x40026760, RQ_count = 0
```

qh Command

Enter the *qh* command to see a stream head read queue. Enter the driver name and minor number to specify the stream head read queue to display. An example is shown below. When *strdb* prints the stream head read queue, you are put in primary mode. This lets you enter navigation commands to look at data structures pointed to by fields in the queue. These navigation commands are described below under “Primary Commands.”

```
qh tmx 0

struct queue 0x584300                                          S:1

q_qinfo   =  0x2944f0   q_pad1[2] = 00
q_first   =  0x0        q_other   =  0x584374
q_last    =  0x0
q_next    =  0x0
q_link    =  0x0
q_ptr     =  0x5f8500
q_count   =  0
q_flag    =  0x1029
   QREADR
   QWANTR
   QUSE
   QSYNCH
q_minpsz  =  0
q_maxpsz  =  -1
q_hiwat   =  0x200
q_lowat   =  0x100
q_bandp   =  0x0
q_nband   =  0
q_pad1[0] = 00
q_pad1[1] = 00
```

Primary Commands

strdb provides two types of primary mode commands. One kind is used to navigate through data structures. The other kind changes the characteristics of the *strdb* session.

Data Structure Navigation Commands

When you enter the *qh* command, *strdb* prints the stream head read queue and puts you in primary mode. You can enter navigation commands to look at data structures pointed to by fields in the queue. Note that primary mode does not prompt you for commands; you just enter the command keys. You do not need to enter a carriage return with navigation commands. In the example below, a ? is entered to see which fields *strdb* can format. *strdb*

prints the commands for formatting these fields. A carriage return will clear the help screen and redisplay the stream head read queue. In the example below, the *m* key is entered to see the message block pointed to by *q_first*. Next, a ? is entered to see which message block fields *strdb* can format.

```
qh tmx 1

struct queue  0x21f7600                                          S:1

q_qinfo    =  0x1f7924   q_pad1[2] = 00
q_first    =  0x2156780  q_other   =  0x21f7600
q_last     =  0x2185800
q_next     =  0x0
q_link     =  0x0
q_ptr      =  0x267be8
q_count    =  22518
q_flag     =  0x1120
    QUSE
    QOLD
    QSYNCH
q_minpsz   =  0
q_maxpsz   =  -1
q_hiwat    =  0x200
q_lowat    =  0x100
q_bandp    =  0x0
q_nband    =  0
q_pad1[0]  = 00
q_pad1[1]  = 00


?

navigation for structure queue
'i'      = q_qinfo (qinit)
'm'      = q_first (msgb)
'z'      = q_last (msgb)
'n'      = q_next (queue)
'l'      = q_link (queue)
'b'      = q_bandp (qband)
'o'      = q_other (queue)

-- Hit any key to continue --


<carriage return>

struct queue  0x21f7600                                          S:1

q_qinfo    =  0x1f7924   q_pad1[2] = 00
q_first    =  0x2156780  q_other   =  0x21f7600
q_last     =  0x2185800
q_next     =  0x0
q_link     =  0x0
q_ptr      =  0x267be8
q_count    =  22518
```

```
q_flag     =  0x1120
    QUSE
    QOLD
    QSYNCH
q_minpsz   =  0
q_maxpsz   =  -1
q_hiwat    =  0x200
q_lowat    =  0x100
q_bandp    =  0x0
q_nband    =  0
q_pad1[0]  = 00
q_pad1[1]  = 00


m

struct msgb 0x2156780                                          S:2

b_next  =  0x204ac00
b_prev  =  0x0
b_cont  =  0x21fb700
b_rptr  =  0x2242bf2
b_wptr  =  0x2242bf2
b_datap =  0x0
b_band  =    0
b_pad1  = 00
b_flag  =  0x0
b_pad2  =  0


?

navigation for structure msgb
'n'      = b_next (msgb)
'p'      = b_prev (msgb)
'm'      = b_rptr (b_rptr)
'c'      = b_cont (msgb)
'd'      = b_datap (datab)

-- Hit any key to continue --
```

strdb provides different navigation commands for each data structure it formats. The navigation commands for all the data structures are shown below.

Queue Navigation

```
'i'      = q_qinfo (qinit)
'm'      = q_first (msgb)
'z'      = q_last (msgb)
'n'      = q_next (queue)
'l'      = q_link (queue)
'b'      = q_bandp (qband)
'o'      = q_other (queue)
```

Qinit Navigation

```
'i'      = qi_minfo (module_info)
's'      = qi_mstat (module_stat)
```

Message Block Navigation

```
'n'      = b_next (msgb)
'p'      = b_prev (msgb)
'm'      = b_rptr (b_rptr)
'c'      = b_cont (msgb)
'd'      = b_datap (datab)
```

Data Block Navigation

```
'd'      = db_f (a__datab)
```

Queue Band Navigation

```
'n'      = qb_next (qband)
'f'      = qb_first (msgb)
'l'      = qb_last (msgb)
```

The following information includes more navigation command examples. The *CTRL-P*, *CTRL-T*, :m, *CTRL-U*, *:b*, and *:x* commands, which are used in conjunction with the navigation commands, are shown with examples.

You can enter *?* to see what navigation keys are available.

```
qh tmx 0

struct queue 0x21f7b00                                          S:1

q_qinfo    =  0x1f7a18    q_pad1[2]  = 00
q_first    =  0x0         q_other    =  0x21f7b74
q_last     =  0x0
q_next     =  0x0
q_link     =  0x0
q_ptr      =  0x21f7a00
q_count    =  0
q_flag     =  0x1029
    QREADR
    QWANTR
    QUSE
    QSYNCH
q_minpsz   =  0
q_maxpsz   =  -1
q_hiwat    =  0x200
q_lowat    =  0x100
q_bandp    =  0x0
q_nband    =  0
q_pad1[0]  = 00
```

```
q_pad1[1]  = 00


?

navigation for structure queue
'i'      = q_qinfo (qinit)
'm'      = q_first (msgb)
'z'      = q_last (msgb)
'n'      = q_next (queue)
'l'      = q_link (queue)
'b'      = q_bandp (qband)
'o'      = q_other (queue)

-- Hit any key to continue --
```

After typing a key to continue, you can enter any of the keys shown in the help text. For example, if you enter *o*, the stream head write queue will be displayed. This is shown below.

```
o

struct queue 0x21f7b74                                          S:2

q_qinfo    =  0x1f7a34   q_pad1[2] = 00
q_first    =  0x0        q_other   = 0x21f7b00
q_last     =  0x0
q_next     =  0x21f7674
q_link     =  0x0
q_ptr      =  0x21f7a00
q_count    =  0
q_flag     =  0x102a
    QNOENB
    QWANTR
    QUSE
    QSYNCH
q_minpsz   =  0
q_maxpsz   =  -1
q_hiwat    =  0x2800
q_lowat    =  0x400
q_bandp    =  0x0
q_nband    =  0
q_pad1[0]  = 00
q_pad1[1]  = 00
```

At this point you can enter *n* to see the next write queue on the stream. Note that *strdb* provides the same navigation keys for each queue structure. Therefore, you can enter the same keys for the stream head write queue as for the stream head read queue. An example of entering the *n* key is shown below.

```
n

struct queue 0x21f7674                                          S:3
```

```
q_qinfo    =  0x1f7924    q_pad1[2] = 00
q_first    =  0x2156780   q_other   =  0x21f7600
q_last     =  0x2185800
q_next     =  0x0
q_link     =  0x0
q_ptr      =  0x0029cc48
q_count    =  22518
q_flag     =  0x1120
    QUSE
    QOLD
    QSYNCH
q_minpsz   =  0
q_maxpsz   =  -1
q_hiwat    =  0x8000
q_lowat    =  0x4000
q_bandp    =  0x0
q_nband    =  0
q_pad1[0]  = 00
q_pad1[1]  = 00
```

This queue contains a non-zero *q_first* pointer. The *m* navigation key can be used to look at the messages on the queue. This is shown below.

```
m

struct msgb 0x2156780                                          S:4

b_next  =  0x204ac00
b_prev  =  0x0
b_cont  =  0x21fb700
b_rptr  =  0x2242bf2
b_wptr  =  0x2242bf2
b_datap =  0x0
b_band  =    0
b_pad1  = 00
b_flag  =  0x0
b_pad2  =  0
```

The *m* key displays the first message on the queue. The *?* command shows the navigation queues available for the message block data structure.

```
?

navigation for structure msgb
'n'      = b_next (msgb)
'p'      = b_prev (msgb)
'm'      = b_rptr (b_rptr)
'c'      = b_cont (msgb)
'd'      = b_datap (datab)

-- Hit any key to continue --
```

The *n* key shows the next message on the queue.

```
n

struct msgb 0x204ac00                                                S:5

b_next  =  0x21f4b00
b_prev  =  0x218ee00
b_cont  =  0x2198080
b_rptr  =  0x223dc00
b_wptr  =  0x223ddc3
b_datap =  0x204ac40
b_band  =     0
b_pad1  = 00
b_flag  =  0x0
b_pad2  =  0
```

The *m* key shows the data associated with this message block.

```
m

struct msgb 0x204ac00        Message data at 0x0223dc00            S:6

0x0223dc00   :    01491800    76777879    7a616263      64656667      |
.I..vwxyzabcdefg
0x0223dc10 :  68696a6b  6c6d6e6f  70717273  74757677  |  hijklmnopqrstuvw
0x0223dc20 :  78797a61  62636465  66676869  6a6b6c6d  |  xyzabcdefghijklm
0x0223dc30 :  6e6f7071  72737475  76777879  7a616263  |  nopqrstuvwxyzabc
0x0223dc40 :  64656667  68696a6b  6c6d6e6f  70717273  |  defghijklmnopqrs
0x0223dc50 :  74757677  78797a61  62636465  66676869  |  tuvwxyzabcdefghi
0x0223dc60 :  6a6b6c6d  6e6f7071  72737475  76777879  |  jklmnopqrstuvwxy
0x0223dc70 :  7a616263  64656667  68696a6b  6c6d6e6f  |  zabcdefghijklmno
0x0223dc80 :  70717273  74757677  78797a61  62636465  |  pqrstuvwxyzabcde
0x0223dc90 :  66676869  6a6b6c6d  6e6f7071  72737475  |  fghijklmnopqrstu
0x0223dca0 :  76777879  7a616263  64656667  68696a6b  |  vwxyzabcdefghijk
0x0223dcb0 :  6c6d6e6f  70717273  74757677  78797a61  |  lmnopqrstuvwxyza
0x0223dcc0 :  62636465  66676869  6a6b6c6d  6e6f7071  |  bcdefghijklmnopq
0x0223dcd0 :  72737475  76777879  7a616263  64656667  |  rstuvwxyzabcdefg
0x0223dce0 :  68696a6b  6c6d6e6f  70717273  74757677  |  hijklmnopqrstuvw
0x0223dcf0 :  78797a61  62636465  66676869  6a6b6c6d  |  xyzabcdefghijklm
0x0223dd00 :  6e6f7071  72737475  76777879  7a616263  |  nopqrstuvwxyzabc
0x0223dd10 :  64656667  68696a6b  6c6d6e6f  70717273  |  defghijklmnopqrs
0x0223dd20 :  74757677  78797a61  62636465  66676869  |  tuvwxyzabcdefghi
0x0223dd30 :  6a6b6c6d  6e6f7071  72737475  76777879  |  jklmnopqrstuvwxy
Type c for more data
Any other key will quit this display
```

You can continue to type the *c* key to see the rest of the data. Enter a key other than *c* to stop examining data.

Note that each time *strdb* displays a data structure, it pushes it onto a stack. *strdb* saves structures on a stack so you can re-examine them later. *strdb* increments and displays the stack depth. The depth appears in the upper right hand corner of the screen as “S:depth.” In the current example, the message data is on the top of the stack, and the depth is 6.

At this point, you may want to see the next message in the queue. To do this, enter a key other than *c* to stop examining data. Then you can enter the primary mode command *CTRL-P* to pop the message data and get back to the message block for the data. This is shown below.

```
<< press a key besides c >>

^P

struct msgb 0x204ac00                                          S:5

b_next  =  0x21f4b00
b_prev  =  0x218ee00
b_cont  =  0x2198080
b_rptr  =  0x223dc00
b_wptr  =  0x223ddc3
b_datap =  0x204ac40
b_band  =    0
b_pad1  = 00
b_flag  =  0x0
b_pad2  =  0
```

In this example, you could have returned to the message block by entering *CTRL-T* to transpose the top two stack entries instead of popping. This has the advantage that the message data is still on the stack in case you want to look at it later. The last example is redone below using *CTRL-T*. Notice that the stack depth for the message block is 6 after transposing instead of 5 after popping.

```
^T

struct msgb 0x204ac00                                          S:6

b_next  =  0x21f4b00
b_prev  =  0x218ee00
b_cont  =  0x2198080
b_rptr  =  0x223dc00
b_wptr  =  0x223ddc3
b_datap =  0x204ac40
b_band  =    0
b_pad1  = 00
b_flag  =  0x0
b_pad2  =  0
```

Besides popping the top of the stack or transposing stack entries, you can pop back to a mark. Enter the *:m* command to set a mark on the data structure stack. Later, enter *CTRL-U* to pop back to the structure with the mark. For example, suppose that in the previous examples *:m* was entered

after *strdb* displayed the write queue below the stream head write queue. Then in the current example, *CTRL-U* could be entered to pop back to this queue. This is shown below.

```
^U

struct queue 0x21f7674                                          S:3

q_qinfo    =  0x1f7924    q_pad1[2] = 00
q_first    =  0x2156780   q_other   =  0x21f7600
q_last     =  0x2185800
q_next     =  0x0
q_link     =  0x0
q_ptr      =  0x0029cc48
q_count    =  22518
q_flag     =  0x1120
    QUSE
    QOLD
    QSYNCH
q_minpsz   =  0
q_maxpsz   =  -1
q_hiwat    =  0x8000
q_lowat    =  0x4000
q_bandp    =  0x0
q_nband    =  0
q_pad1[0]  = 00
q_pad1[1]  = 00
```

When you enter the *CTRL-U* command, *strdb* prints the data it saved in the marked entry. If you are running *strdb* on a running system instead of a core file, the data may not be current. In the above example, the queue may contain different data when *CTRL-U* is entered than it did when the contents of the queue were pushed on the stack. To see the current values, enter the *CTRL-R* command. *CTRL-R* updates the displayed data structure with new values from */dev/kmem*. This is shown below. Notice that there are no longer any messages in the queue.

```
^R

struct queue 0x21f7674                                          S:3

q_qinfo    =  0x1f7924    q_pad1[2] = 00
q_first    =  0x0           q_other   =  0x21f7600
q_last     =  0x0
q_next     =  0x0
q_link     =  0x0
q_ptr      =  0x0029cc48
q_count    =  0
q_flag     =  0x1120
    QUSE
    QOLD
    QSYNCH
```

```
q_minpsz   =  0
q_maxpsz   =  -1
q_hiwat    =  0x8000
q_lowat    =  0x4000
q_bandp    =  0x0
q_nband    =  0
q_pad1[0]  = 00
q_pad1[1]  = 00
```

You may want to use *CTRL-R* when you are entering navigation commands, not just when you pop the data structure stack. This is because *strdb* does not automatically update the display when the contents of data structures change. You need to enter the *CTRL-R* command to update the display with new values from */dev/kmem*.

In the previous example, suppose you want to print a field in the queue that *strdb* does not format. This can be done using *:b*. The *:b* command prints the contents of memory starting at a specified address. Optionally, you can specify the number of bytes that *:b* should print. If you want to see the *q_ptr* structure, enter the following.

```
:b 0x29cc48

 0x0029cc48 : 00000001  005d9a00  00000000  00000000  |
.....]..........
 0x0029cc58 : 00000001  005d8b00  00000000  00000000  |  .....]..........
 0x0029cc68 : 00000001  00605100  00000000  00000000  |  .....`Q.........
 0x0029cc78 : 00000000  00000000  00000000  00000000  |  ................
 0x0029cc88 : 00000000  00000000  00000000  00000000  |  ................
 0x0029cc98 : 00000000  00000000  00000000  00000000  |  ................
 0x0029cca8 : 00000000  00000000  00000000  00000000  |  ................
 0x0029ccb8 : 00000000  00000000  00000000  00000000  |  ................
 0x0029ccc8 : 00000000  00000000  00000000  00000000  |  ................
 0x0029ccd8 : 00000000  00000000  00000000  00000000  |  ................
 0x0029cce8 : 00000000  00000000  00000000  00000000  |  ................
 0x0029ccf8 : 00000000  00000000  00000000  00000000  |  ................
 0x0029cd08 : 00000000  00000000  00000000  00000000  |  ................
 0x0029cd18 : 00000000  00000000  00000000  00000000  |  ................
 0x0029cd28 : 00000000  00000000  00000000  00000000  |  ................
 0x0029cd38 : 00000000  00000000  00000000  00000000  |  ................

-- Hit any key to continue --
```

The *:x* command is often used with *:b*. If the *q_ptr* buffer contains a pointer to a STREAMS/UX data structure, you can format the structure using *:x*. You know that word 0x0029cc4c in the *q_ptr* buffer contains a queue address, 0x005d9a00. The *:x* command takes two arguments, a structure address and its type. You can enter *:x ?* to see which types are accepted by the *:x* command. This is shown below.

```
:x ?

known data structure descriptions...
   streamtab
   msgb
   a__datab
   datab
   free_rtn
   queue
   qband
   qinit
   module_info
   module_stat
   strapush
   ioc_pad
   iocblk
   copyreq
   copyresp
   stroptions

-- Hit any key to continue --
```

The type for a STREAMS/UX queue is *queue*. You can double check which type to use by looking in the include file <*sys/stream.h*>. An example of entering the *:x* command to format the queue is shown below.

```
:x queue 0x5d9a00

struct queue 0x5d9a00                                          S:4

q_qinfo   =  0x294418    q_pad1[1] = 00
q_first   =  0x0         q_pad1[2] = 00
q_last    =  0x0         q_other   =  0x5d9a74
q_next    =  0x5ceb00
q_link    =  0x0
q_ptr     =  0x29cc48
q_count   =  0
q_flag    =  0x1129
   QREADR
   QWANTR
   QUSE
   QOLD
   QSYNCH
q_minpsz  =  0
q_maxpsz  =  256
q_hiwat   =  0x8000
q_lowat   =  0x4000
q_bandp   =  0x5393c0
q_nband   =  1
q_pad1[0] = 00
```

Commands to Change strdb Session Characteristics

After entering *strdb*, you can enter the *:?* command to get information about primary commands. Note that primary mode does not prompt for commands; you just enter the command keys.

```
:?

key                - navigate from current structure
^D | :q            - exit
^L                 - refresh
^K                 - log screen contents if logging enabled
?                  - show navigation keys for current structure
:?                 - show known commands
:x ?               - show known structure descriptions
:x 'name' 'addr'   - show structure 'name' at address 'addr'
:b 'addr' 'len'    - show screenful of binary data at address
'addr'
                     ('len' defaults to 256 if not specified)
^P                 - pop stack
^U                 - pop stack to previous mark
^T                 - transpose top stack entries
^R                 - re-read current structure from memory
:s                 - enable structure Stacking
:l 'name' 'o|c'    - start[o] / stop[c] logging to 'name'
:m                 - mark current stack location
:u                 - Unenable structure stacking
:S                 - STREAMS subsystem commands
```

There are two types of primary commands, data structure navigation and commands to change *strdb* session characteristics. This section describes the commands that change *strdb* session characteristics:

- :?
- CTRL-D
- :q
- CTRL-L
- :l
- :u
- :s
- :S

Enter the *:?* command to see the help menu for primary mode. *strdb* prints the text shown below.

```
:?

key                - navigate from current structure
^D | :q            - exit
^L                 - refresh
^K                 - log screen contents if logging enabled
?                  - show navigation keys for current structure
:?                 - show known commands
:x ?               - show known structure descriptions
:x 'name' 'addr'   - show structure 'name' at address 'addr'
:b 'addr' 'len'    - show screenful of binary data at address
'addr'
                     ('len' defaults to 256 if not specified)
^P                 - pop stack
^U                 - pop stack to previous mark
^T                 - transpose top stack entries
^R                 - re-read current structure from memory
:s                 - enable structure Stacking
:l 'name' 'o|c'    - start[o] / stop[c] logging to 'name'
:m                 - mark current stack location
:u                 - Unenable structure stacking
:S                 - STREAMS subsystem commands
```

Enter the *CTRL-D* or the *:q* command to exit from *strdb*.

Enter the *CTRL-L* command to refresh the screen.

Enter the *:l* command to start and stop logging to a file. *strdb* will log commands and their output to a file. Enter the *:l* command specifying a file name and the o option to open the log file and start logging. Then you can enter *strdb* commands and see the output on the terminal. *strdb* saves a record of the commands and output in the logging file. Once logging is enabled, use *CTRL-K* to dump the current screen contents to the log file. This allows the user to selectively log debug data and actions taken. You can close the log file and stop logging by entering the *:l* command, the file name, and the *c* option. An example is shown below.

```
:l strdb.log o

                    No current structure                    S:0

:S

STREAMS subsystem help commands..
?                  - show this help menu
h                  - show this help menu
la  'name'         - list all active STREAMS on device 'name'
ll  'name' 'minor' - list all drivers linked under the STREAMS
driver
```

```
                    'name' and minor number 'minor'
lm  'name' 'minor'  - list all modules pushed on STREAMS device
                    'name' and whose minor number is 'minor'
lp  'name' 'minor'  - list all drivers persistently linked under
                    the STREAMS device 'name' and minor number
                    'minor'
q                 - quit the STREAMS subsystem commands
qc 'driver' 'file'  - print 'driver' read / write side qcount to
                    file
qh  'name' 'minor' - display STREAM head queue structure
                     for device 'name' and minor number
                    'minor'
s   [m | d]         - Option d lists all the STREAMS drivers
                      configured in the system. Option m lists
                      all the modules configured in the system
v                 - print version of STREAMS structures
                     displayed

qh tmx 1

struct queue 0x20a2300                                      S:1

q_qinfo    =  0x1f7a18   q_pad1[2] = 00
q_first    =  0x0        q_other   =  0x20a2374
q_last     =  0x0
q_next     =  0x0
q_link     =  0x0
q_ptr      =  0x206d900
q_count    =  0
q_flag     =  0x1029
    QREADR
    QWANTR
    QUSE
    QSYNCH
q_minpsz   =  0
q_maxpsz   =  -1
q_hiwat    =  0x200
q_lowat    =  0x100
q_bandp    =  0x0
q_nband    =  0
q_pad1[0]  = 00
q_pad1[1]  = 00

: ^k (screen data is dumped to strdb.log)

:l strdb.log c

:u and :s
```

When you enter *strdb*, data structure stacking is enabled. Each time *strdb* displays a data structure, it pushes it onto a stack. *strdb* increments and displays the stack depth. Data structure stacking is useful for going back and reviewing data structures that *strdb* has already displayed. This is described in the previous section, “Data Structure Navigation Commands.”

You can disable data structure stacking by entering the *:u* command. When data structure stacking is disabled, strdb does not display the stack depth. Data structure stacking is re-enabled by entering the *:s* command. An example is shown below. Note how the stack depth displayed in the upper right hand corner of the screen changes.

```
qh tmx 0

struct queue 0x21f7b00                                          S:1

q_qinfo    =  0x1f7a18    q_pad1[2]  = 00
q_first    =  0x0         q_other    =  0x21f7b74
q_last     =  0x0
q_next     =  0x0
q_link     =  0x0
q_ptr      =  0x21f7a00
q_count    =  0
q_flag     =  0x1029
    QREADR
    QWANTR
    QUSE
    QSYNCH
q_minpsz   =  0
q_maxpsz   =  -1
q_hiwat    =  0x200
q_lowat    =  0x100
q_bandp    =  0x0
q_nband    =  0
q_pad1[0]  = 00
q_pad1[1]  = 00


:u

struct queue 0x21f7b00

q_qinfo    =  0x1f7a18    q_pad1[2]  = 00
q_first    =  0x0         q_other    =  0x21f7b74
q_last     =  0x0
q_next     =  0x0
q_link     =  0x0
q_ptr      =  0x21f7a00
q_count    =  0
q_flag     =  0x1029
    QREADR
    QWANTR
    QUSE
    QSYNCH
q_minpsz   =  0
q_maxpsz   =  -1
q_hiwat    =  0x200
q_lowat    =  0x100
q_bandp    =  0x0
q_nband    =  0
```

```
q_pad1[0]  = 00
q_pad1[1]  = 00


o

struct queue 0x21f7b74

q_qinfo    =  0x1f7a34   q_pad1[2] = 00
q_first    =  0x0        q_other   = 0x21f7b00
q_last     =  0x0
q_next     =  0x21f7674
q_link     =  0x0
q_ptr      =  0x21f7a00
q_count    =  0
q_flag     =  0x102a
    QNOENB
    QWANTR
    QUSE
    QSYNCH
q_minpsz   =  0
q_maxpsz   =  -1
q_hiwat    =  0x2800
q_lowat    =  0x400
q_bandp    =  0x0
q_nband    =  0
q_pad1[0]  = 00
q_pad1[1]  = 00


:s

struct queue 0x21f7b74                                          S:1

q_qinfo    =  0x1f7a34   q_pad1[2] = 00
q_first    =  0x0        q_other   = 0x21f7b00
q_last     =  0x0
q_next     =  0x21f7674
q_link     =  0x0
q_ptr      =  0x21f7a00
q_count    =  0
q_flag     =  0x102a
    QNOENB
    QWANTR
    QUSE
    QSYNCH
q_minpsz   =  0
q_maxpsz   =  -1
q_hiwat    =  0x2800
q_lowat    =  0x400
q_bandp    =  0x0
q_nband    =  0
q_pad1[0]  = 00
q_pad1[1]  = 00

:S
```

Enter the *:S* command to switch from primary mode to STREAMS/UX subsystem mode. After invoking *strdb*, you are in primary mode. Enter *:S* to switch to STREAMS/UX subsystem mode. In STREAMS/UX subsystem mode, you can see which STREAMS/UX are configured and active on the system. An example is shown below.

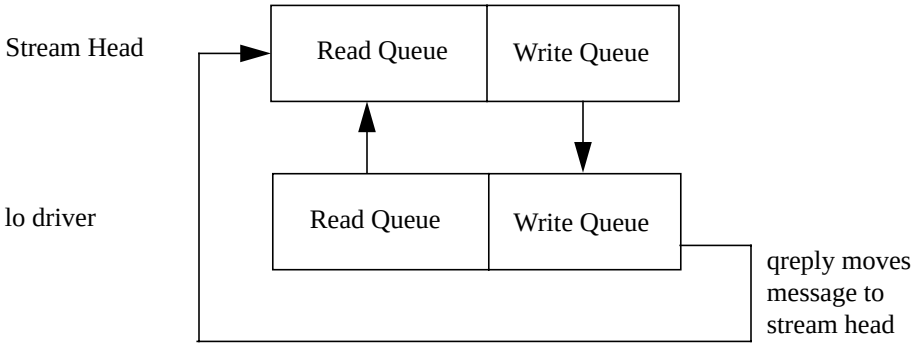
```
strdb

                     No current structure                  S:0

:S


STREAMS subsystem help commands..
?                   - show this help menu
h                   - show this help menu
la  'name'          - list all active STREAMS on device 'name'
ll  'name' 'minor' - list all drivers linked under the STREAMS
                      driver 'name' and minor number 'minor'
lm  'name' 'minor' - list all modules pushed on STREAMS device
                      'name' and whose minor number is 'minor'
lp  'name' 'minor'  - list all drivers persistently linked under
the
                      STREAMS device 'name' and minor number
'minor'
q                   - quit the STREAMS subsystem commands
qc 'driver' 'file'  - print 'driver' read / write side qcount
to file
qh  'name' 'minor' - display STREAM head queue structure
                    for device 'name' and minor number 'minor'
s   [m | d]         - Option d lists all the STREAMS drivers
                      configured in the system. Option m lists
                      all the modules configured in the system
v                   - print version of STREAMS structures
displayed
```

Debugging with strdb

This section shows examples of using *strdb* to debug STREAMS/UX drivers and modules. The examples show how to use *strdb* on a running system. The *adb* debugging section of this chapter shows an example of using *strdb* in conjunction with *adb* to analyze an HP-UX core file.

Example 1: Flow Control and Fragmentation

In this example, the user has written a loopback driver which uses the *qreply* STREAMS/UX utility to send all incoming messages up to the stream head read queue.




```
lo_write_put(q,m) ;
 if m not hipri
     putq
 else
     qreply
```

```
lo_write_srv(q) ;
  if stream head not flow controlled
    qreply
  else
    putbq
```

Figure 2 **Stream Created By Opening Loopback (lo) Driver**

The user writes a simple test for the driver. The test opens *lo*, writes data to it, reads the data, and then closes the driver. The program is shown below.

```
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>

main()
{

   char         wbuf[1024];
   char         rbuf[1024];
   int          fd, i, n, cnt;

   printf("Open the loopback driver.\n");
        fd = open("/dev/lo0", O_RDWR);
```

```
        if (fd < 0)
           printf("Open returned %d and errno = %d.\n",
                  n, errno);

   /* Fill buffer with data to write */
   for (n = 0; n < 1024; n++)
         wbuf[n] = (char) n;

   printf("Call write with nbytes set to 1024.\n");
         n = write(fd, wbuf, 1024);
         if (n != 1024)
            printf("Write returned %d and errno = %d.\n",
                   n, errno);

   printf("Call read to read in the message sent down
          stream.\n");
         n = read(fd, rbuf, 1024);
         if (n != 1024)
            printf("Read returned %d and errno =
                   %d.\n",n,errno);

   printf("Close the loopback driver.\n");
         close( fd );


}
```

When the user runs the program, it prints the following results:

```
Open the loopback driver.
Call write with nbytes set to 1024.
Call read to read in the message sent down stream.
Read returned 512 and errno = 0.
Close the loopback driver.
```

The user runs *strdb* to find out why the test program read only 512 bytes of data instead of 1024. First, the user changes the test program to sleep between the *write()* and *read()* calls. When the program sleeps, the user runs *strdb* to see what happened to the data. This is shown below.

```
strdb

                        No current structure                S:0
```

The user types *:S* to enter STREAMS/UX subsystem mode.

```
:S

STREAMS subsystem help commands..
?                  - show this help menu
d                  - print status of STREAMS daemon
h                  - show this help menu
la  'name'         - list all active STREAMS on device 'name'
ll  'name' 'minor' - list all drivers linked under the STREAMS
driver
```

```
                    'name' and minor number 'minor'
lm  'name' 'minor' - list all modules pushed on STREAMS device
                    'name' and whose minor number is 'minor'
lp  'name' 'minor' - list all drivers persistently linked under
                    the STREAMS device 'name' and minor number
                    'minor'
q                  - quit the STREAMS subsystem commands
qc 'driver' 'file'  - print 'driver' read / write side qcount
                    to file
qh  'name' 'minor' - display STREAM head queue structure
                     for device 'name' and minor number
                     'minor'
s   [m | d]         - Option d lists all the STREAMS drivers
                      configured in the system. Option m lists
                      all the modules configured in the system
v                 - print version of STREAMS structures
                    displayed
```

Then the user enters the *la* command for *lo* to see what minor number the driver assigned to the stream.

```
la  lo

                          stack empty                          S:0

lo MAJOR = 75
ACTIVE Minor 0x000000  Stream head RQ = 0x00515500


-- Hit any key to continue --
```

Next, the user enters *qh* for *lo* and minor number 0 to start examining the stream. strdb formats the stream head read queue.

```
qh lo 0


struct queue 0x515500                                          S:1

q_qinfo    =  0x2954f0   q_pad1[2] = 00
q_first    =  0x50da00   q_other   =  0x515574
q_last     =  0x513780
q_next     =  0x0
q_link     =  0x0
q_ptr      =  0x530600
q_count    =  512
q_flag     =  0x103d
   QREADR
   QFULL
   QWANTR
   QWANTW
   QUSE
   QSYNCH
q_minpsz   =  0
q_maxpsz   =  -1
```

```
q_hiwat    =  0x200
q_lowat    =  0x100
q_bandp    =  0x0
q_nband    =  0
q_pad1[0]  = 00
q_pad1[1]  = 00
```

The user notes that *q_count*, the number of bytes of data on the queue, is 512. This is the amount of data the test program was able to read. The user realizes that the test program could only read 512 bytes, because that is all that was in the queue. The user continues examining the stream in order to find out what happened to the other 512 bytes of data. The user enters the *o* navigation key to see the other queue, the stream head write queue.

```
o

struct queue 0x515574                                          S:2

q_qinfo    =  0x29550c   q_pad1[2] = 00
q_first    =  0x0        q_other   =  0x515500
q_last     =  0x0
q_next     =  0x4bc974
q_link     =  0x0
q_ptr      =  0x530600
q_count    =  0
q_flag     =  0x102a
   QNOENB
   QWANTR
   QUSE
   QSYNCH
q_minpsz   =  0
q_maxpsz   =  -1
q_hiwat    =  0x2800
q_lowat    =  0x400
q_bandp    =  0x0
q_nband    =  0
q_pad1[0]  = 00
q_pad1[1]  = 00
```

The user sees that there is no data in this queue. The user enters the *n* key to see the next queue, *lo*'s write queue.

```
n

struct queue 0x4bc974                                          S:3

q_qinfo    =  0x2951cc   q_pad1[2] = 00
q_first    =  0x537800   q_other   =  0x4bc900
q_last     =  0x50d100
q_next     =  0x0
q_link     =  0x0
q_ptr      =  0x2b1fa8
q_count    =  512
q_flag     =  0x1124
```

```
   QFULL
   QUSE
   QOLD
   QSYNCH
q_minpsz   =  0
q_maxpsz   =  256
q_hiwat    =  0x200
q_lowat    =  0x100
q_bandp    =  0x0
q_nband    =  0
q_pad1[0]  = 00
q_pad1[1]  = 00
```

The user sees that the rest of the data is on this queue. The user wonders why the *lo* driver did not put this data on the stream head write queue. The user enters the *CTRL-P* command to go back to the stream head read queue.

```
^P

struct queue 0x515574                                          S:2

q_qinfo    =  0x29550c   q_pad1[2] = 00
q_first    =  0x0        q_other   = 0x515500
q_last     =  0x0
q_next     =  0x4bc974
q_link     =  0x0
q_ptr      =  0x530600
q_count    =  0
q_flag     =  0x102a
   QNOENB
   QWANTR
   QUSE
   QSYNCH
q_minpsz   =  0
q_maxpsz   =  -1
q_hiwat    =  0x2800
q_lowat    =  0x400
q_bandp    =  0x0
q_nband    =  0
q_pad1[0]  = 00
q_pad1[1]  = 00


^P

struct queue 0x515500                                          S:1

q_qinfo    =  0x2954f0   q_pad1[2] = 00
q_first    =  0x50da00   q_other   = 0x515574
q_last     =  0x513780
q_next     =  0x0
q_link     =  0x0
q_ptr      =  0x530600
q_count    =  512
```

```
q_flag     =  0x103d
   QREADR
   QFULL
   QWANTR
   QWANTW
   QUSE
   QSYNCH
q_minpsz   =  0
q_maxpsz   =  -1
q_hiwat    =  0x200
q_lowat    =  0x100
q_bandp    =  0x0
q_nband    =  0
q_pad1[0]  = 00
q_pad1[1]  = 00
```

The user notices that the QFULL flag is set. This indicates that the queue is flow controlled. *q_hiwat* is set to 0x200 (512 decimal). Therefore, *lo* can write only 512 bytes of data to the stream head before a user program does a read, relieving the flow control condition.

The user realizes that this problem occurs because STREAMS/UX fragmented the 1024 bytes into smaller messages. If STREAMS/UX put all the data in one message, *lo* would put the entire message on the stream head read queue. *lo* would be able to do this because the driver tests once for flow control before sending the data upstream. Then, when *lo* tests for flow control, the stream head read queue is empty. *lo* cannot send all the data when it is fragmented because *lo* must check for flow control before sending each fragment. After 512 bytes are in the stream head write queue, the flow control check fails.

The user wonders why STREAMS/UX fragmented the data. The user enters *m* to look at the fragments.

```
m

struct msgb 0x50da00                                          S:2

b_next  =  0x513780
b_prev  =  0x0
b_cont  =  0x0
b_rptr  =  0x47a700
b_wptr  =  0x47a800
b_datap =  0x50da40
b_band  =    0
b_pad1  = 00
b_flag  =  0x0
b_pad2  =  0
```

The user notes that there are 256 bytes in this message (*b_wptr* - *b_rptr* = 256). The user looks at the next message by entering the *n* key.

```
n

struct msgb 0x513780                                              S:3

b_next  =  0x0
b_prev  =  0x50da00
b_cont  =  0x0
b_rptr  =  0x4efc00
b_wptr  =  0x4efd00
b_datap =  0x5137c0
b_band  =    0
b_pad1  = 00
b_flag  =  0x0
b_pad2  =  0
```

This message also contains 256 bytes. The user enters navigation commands to view *lo*'s write queue. The user examines the sizes of the messages on this queue. They are also 256 bytes. The user reads documentation describing how STREAMS/UX executes the *write()* system call. According to the stream(2) man page, STREAMS/UX fragments when the data size is larger than the topmost stream module's *maxpsz*. *lo* is the topmost stream module; its *maxpsz* is 256.

The user can fix this problem in two ways. One way is to change the test program to perform multiple reads to receive all the data. Another way is to change the driver's *maxpsz* to be 1024.

Example 2: Simple Driver Programming Error

In this example, the user has written a loopback driver, *sp*, which uses timeout to simulate interrupts. *sp*'s put routine calls *timeout* for each message it receives. When the timeout expires, HP-UX calls *sp*'s timeout function. This function calls *putq()* to put the message on *sp*'s read queue.

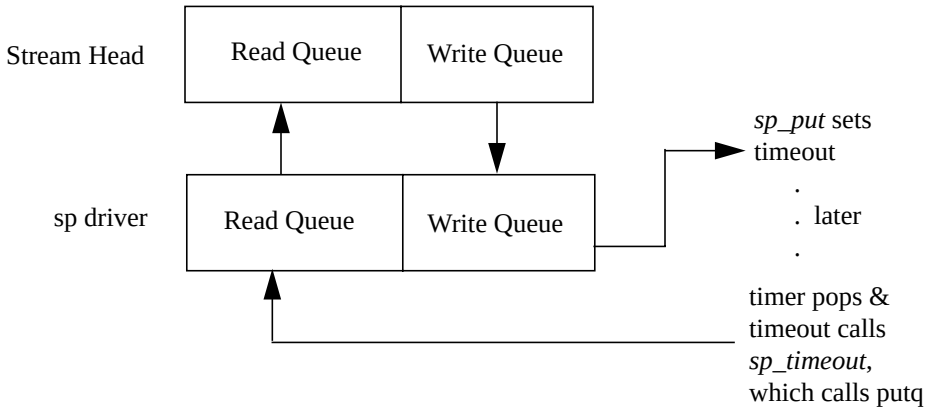


Figure 3 **Stream Created By Opening Loopback (sp) Driver**

The *sp_put()* routine puts the incoming message on a queue in *sp*'s private data structure before calling *timeout()*. *sp*'s timeout function takes the first message off the queue, and calls *putq* to put the message on *sp*'s read queue. *sp*'s open routine saves a pointer to *sp*'s private data structure in the write and read queues' *q_ptr* field. *sp*'s private data structure and the *sp_put()* and *sp_timeout()* routines are shown below.

```
struct sp {
  unsigned sp_state;   /* Set to SPOPEN when driver opened. */
                       /* Cleared when driver is closed.    */
  queue_t  *sp_rdq;    /* Contains sp's read q pointer.     */
  mblk_t   *first_mp;  /* Pointer to head of message list.  */
                       /* Messages are saved here until     */
                       /* timeout expires.                  */
  mblk_t   *last_mp;   /* Pointer to tail of message list.  */
};


/* Driver state values. */
#define SPOPEN 01



static sp_put(q, mp)
queue_t *q;
mblk_t *mp;
{
  struct sp *private;
  unsigned int s;

  /*
   * Check the message type.
   */
  switch (mp->b_datap->db_type) {
  case M_DATA:
  case M_PROTO:
  case M_PCPROTO:
    /* Raise the spl level to protect private structure,
     * since timeout functions such as sp_timeout can
     * interrupt sp_put.
     */
    s = splstr();
       /* Put the message at the tail of the
        * private data structure queue.
        */
       private = q->q_ptr;
       if (!private->last_mp)
         private->first_mp = mp;
       else
         private->last_mp->b_next = mp;
       private->last_mp = mp;
    splx(s);
    /* Set the timeout */
    timeout(sp_timeout,private,1);
    break;
  default:
    printf("Routine sp_put: Illegal message %x received.\n",
           mp->b_datap->db_type);
    break;
  }
}
```

```
static sp_timeout(private)
struct sp *private;
{
  mblk_t *temp;
  unsigned int s;

  /* Make sure driver isn't being closed. */
  if ((private->sp_state & SPOPEN) && (private->first_mp)) {
        /* Take message off head of queue in private data
structure. */
      temp = private->first_mp;
      private->first_mp = private->first_mp->b_next;
      temp->b_next = NULL;
       /* Call putq to put message on sp's read queue and send
it          upstream. */
      putq(private->sp_rdq, temp);
  }
}
```

The user writes a test for the driver. The test opens *sp*, and goes into a loop calling *putmsg()* to send data and calling *getmsg()* to receive the data back. The test prints a message each time it receives 100 messages. The user runs the program, but it does not print any messages. While the program is running, the user runs *strdb* to see what is happening on the stream. This is shown below.

```
strdb

                          No current structure                S:0
```

The user types *:S* to enter STREAMS/UX subsystem mode.

```
:S

STREAMS subsystem help commands..
?                 - show this help menu
d                 - print status of STREAMS daemon
h                 - show this help menu
la  'name'        - list all active STREAMS on device 'name'
ll  'name' 'minor' - list all drivers linked under the STREAMS
                     driver 'name' and minor number 'minor'
lm  'name' 'minor' - list all modules pushed on STREAMS device
                      'name' and whose minor number is 'minor'
lp  'name' 'minor' - list all drivers persistently linked under
                      the STREAMS device 'name' and minor number
                     'minor'
q                 - quit the STREAMS subsystem commands
qc 'driver' 'file' - print 'driver' read / write side qcount to
file
qh  'name' 'minor'- display STREAM head queue structure
                     for device 'name' and minor number 'minor'
s   [m | d]       - Option d lists all the STREAMS drivers
                    configured in the system. Option m lists
```

```
                    all the modules configured in the system
v                - print version of STREAMS structures
displayed
```

Then the user enters the *la* command for *sp* to see what minor number the driver assigned to the stream.

```
la  sp

                          stack empty                          S:0

sp MAJOR = 115
ACTIVE Minor 0x000000  Stream head RQ = 0x005c1500


-- Hit any key to continue --
```

Next, the user enters the *qh* command for *sp* and minor number 0 to start examining the stream. *strdb* formats the stream head read queue.

```
qh sp 0

struct queue 0x5c1500                                          S:1

q_qinfo    =  0x2964f0   q_pad1[2] = 00
q_first    =  0x0        q_other   =  0x5c1574
q_last     =  0x0
q_next     =  0x0
q_link     =  0x0
q_ptr      =  0x5f0100
q_count    =  0
q_flag     =  0x1029
   QREADR
   QWANTR
   QUSE
   QSYNCH
q_minpsz   =  0
q_maxpsz   =  -1
q_hiwat    =  0x200
q_lowat    =  0x100
q_bandp    =  0x0
q_nband    =  0
q_pad1[0]  = 00
q_pad1[1]  = 00
```

The user sees that there are no messages on the stream head read queue. The user decides to look for messages on other queues in the stream. The user enters the *o* key to see the other queue in this pair, the stream head write queue.

```
o

struct queue 0x5c1574                                          S:2

q_qinfo    =  0x29650c   q_pad1[2] = 00
q_first    =  0x0        q_other   =  0x5c1500
q_last     =  0x0
q_next     =  0x605e74
q_link     =  0x0
q_ptr      =  0x5f0100
q_count    =  0
q_flag     =  0x102a
   QNOENB
   QWANTR
   QUSE
   QSYNCH
q_minpsz   =  0
q_maxpsz   =  -1
q_hiwat    =  0x2800
q_lowat    =  0x400
q_bandp    =  0x0
q_nband    =  0
q_pad1[0]  = 00
q_pad1[1]  = 00
```

The user looks at the next queue, *sp*'s write queue, by entering the *n* key.

```
struct queue 0x605e74                                          S:3

q_qinfo    =  0x296434   q_pad1[2] = 00
q_first    =  0x0        q_other   =  0x605e00
q_last     =  0x0
q_next     =  0x0
q_link     =  0x0
q_ptr      =  0x29ec48
q_count    =  0
q_flag     =  0x1128
   QWANTR
   QUSE
   QOLD
   QSYNCH
q_minpsz   =  0
q_maxpsz   =  256
q_hiwat    =  0x8000
q_lowat    =  0x4000
q_bandp    =  0x53b400
q_nband    =  1
q_pad1[0]  = 00
q_pad1[1]  = 00
```

Next the user enters the *o* key to look at the other queue in this pair, *sp*'s read queue.

```
o

struct queue 0x605e00                                          S:5

q_qinfo    =  0x296418   q_pad1[2] = 00
q_first    =  0x0        q_other   =  0x605e74
q_last     =  0x0
q_next     =  0x5c1500
q_link     =  0x0
q_ptr      =  0x29ec48
q_count    =  0
q_flag     =  0x1129
   QREADR
   QWANTR
   QUSE
   QOLD
   QSYNCH
q_minpsz   =  0
q_maxpsz   =  256
q_hiwat    =  0x8000
q_lowat    =  0x4000
q_bandp    =  0x53b3c0
q_nband    =  1
q_pad1[0]  = 00
q_pad1[1]  = 00
```

The user sees that there are no messages on the stream. Next, the user examines *sp*'s private data structure. The user enters the *:b* command, specifying the *q_ptr* field value, 0x29ec48.

```
:b 0x29ec48

0x0029ec48 :  00000001  00605e00  00000000  005fb600  | .....`^......uq.
0x0029ec58 :  00000000  00000000  00000000  00000000  | ................
0x0029ec68 :  00000000  00000000  00000000  00000000  | ................
0x0029ec78 :  00000000  00000000  00000000  00000000  | ................
0x0029ec88 :  00000000  00000000  00000000  00000000  | ................
0x0029ec98 :  00000000  00000000  00000000  00000000  | ................
0x0029eca8 :  00000000  00000000  00000000  00000000  | ................
0x0029ecb8 :  00000000  00000000  00000000  00000000  | ................
0x0029ecc8 :  00000000  00000000  00000000  00000000  | ................
0x0029ecd8 :  00000000  00000000  00000000  00000000  | ................
0x0029ece8 :  00000000  00000000  00000000  00000000  | ................
0x0029ecf8 :  00000000  00000000  00000000  00000000  | ................
0x0029ed08 :  00000000  00000000  00000000  00000000  | ................
0x0029ed18 :  00000000  00000000  00000000  00000000  | ................
0x0029ed28 :  00000000  00000000  00000000  00000000  | ................
0x0029ed38 :  00000000  00000000  00000000  00000000  | ................

-- Hit any key to continue --
```

The user sees that the first word of *sp*'s private data structure is 0x00000001. Looking at the *sp* structure declaration shown above, this word is *sp*'s state. The driver is SPOPEN. The next word of *sp*'s private structure is 0x00605e00. According to the *sp* struct declaration, this is *sp*'s read queue

address. As shown above, *strdb* also reports that *sp*'s read queue address is 0x00605e00. The next two words are pointers to messages being saved until timeouts expire. The first word is the head of the message queue. Its value is 0x00000000. The second word is the tail. Its value is 0x005fb600. The user does not understand how the head of the list can be 0 and the tail non-zero. The user decides to ask *strdb* to format the message on the tail of the queue using the *:x* command. First, the user enters the *:x ?* command to see the names of the structures that *strdb* formats.

```
:x ?

known data structure descriptions...
   streamtab
   msgb
   a__datab
   datab
   free_rtn
   queue
   qband
   qinit
   module_info
   module_stat
   strapush
   ioc_pad
   iocblk
   copyreq
   copyresp
   stroptions

-- Hit any key to continue --
```

The user sees that *strdb* formats msgb, a message block. The user can double check that this is the correct structure name by looking in the *sys/stream.h* include file. Then, the user enters the *:x* command to see the message block.

```
:x msgb 0x005fb600


struct msgb 0x5fb600                                          S:6

b_next  =  0x5fb700
b_prev  =  0x0
b_cont  =  0x5fb680
b_rptr  =  0x599400
b_wptr  =  0x5996ac
b_datap =  0x5fb640
b_band  =    0
b_pad1  = 00
b_flag  =  0x0
b_pad2  =  0
```

The user wonders if this is a valid message block. The fields seem to contain correct values. The user checks the data by entering the *m* key.

```
m

struct msgb 0x5fb600        Message data at 0x00599400            S:7

0x00599400 :  00000000  00000000  00000005  00000000  | ................
0x00599410 :  00000294  0000070c  6d6e6f70  71727374  | ........mnopqrst
0x00599420 :  75767778  797a6162  63646566  6768696a  | uvwxyzabcdefghij
0x00599430 :  6b6c6d6e  6f707172  73747576  7778797a  | klmnopqrstuvwxyz
0x00599440 :  61626364  65666768  696a6b6c  6d6e6f70  | abcdefghijklmnop
0x00599450 :  71727374  75767778  797a6162  63646566  | qrstuvwxyzabcdef
0x00599460 :  6768696a  6b6c6d6e  6f707172  73747576  | ghijklmnopqrstuv
0x00599470 :  7778797a  61626364  65666768  696a6b6c  | wxyzabcdefghijkl
0x00599480 :  6d6e6f70  71727374  75767778  797a6162  | mnopqrstuvwxyzab
0x00599490 :  63646566  6768696a  6b6c6d6e  6f707172  | cdefghijklmnopqr
0x005994a0 :  73747576  7778797a  61626364  65666768  | stuvwxyzabcdefgh
0x005994b0 :  696a6b6c  6d6e6f70  71727374  75767778  | ijklmnopqrstuvwx
0x005994c0 :  797a6162  63646566  6768696a  6b6c6d6e  | yzabcdefghijklmn
0x005994d0 :  6f707172  73747576  7778797a  61626364  | opqrstuvwxyzabcd
0x005994e0 :  65666768  696a6b6c  6d6e6f70  71727374  | efghijklmnopqrst
Type c for more data
Any other key will quit this display
```

The user knows that this is the data the test program sends. The user wonders what is in the next message. To see the next message, the user enters a key other than *c* to stop viewing data. Then, the user pops back to the data's message block.

```
^P

struct msgb 0x5fb600                                              S:6

b_next  =  0x5fb700
b_prev  =  0x0
b_cont  =  0x5fb680
b_rptr  =  0x599400
b_wptr  =  0x5996ac
b_datap =  0x5fb640
b_band  =    0
b_pad1  = 00
b_flag  =  0x0
b_pad2  =  0
```

Next the user enters the *n* key to see the next message block.

```
n


struct msgb 0x5fb700                                              S:7

b_next  =  0x5fb800
b_prev  =  0x0
b_cont  =  0x5fb780
b_rptr  =  0x599800
```

```
b_wptr  =  0x599b36
b_datap =  0x5fb740
b_band  =    0
b_pad1  = 00
b_flag  =  0x0
b_pad2  =  0
```

Again, the values in this message block appear valid. The user double checks the data by entering the *m* key.

```
m

struct msgb 0x5fb700        Message data at 0x00599800            S:8

0x00599800 :  00000000  00000000  00000006  00000000  | ................
0x00599810 :  0000031e  0000081e  7778797a  61626364  | ........wxyzabcd
0x00599820 :  65666768  696a6b6c  6d6e6f70  71727374  | efghijklmnopqrst
0x00599830 :  75767778  797a6162  63646566  6768696a  | uvwxyzabcdefghij
0x00599840 :  6b6c6d6e  6f707172  73747576  7778797a  | klmnopqrstuvwxyz
0x00599850 :  61626364  65666768  696a6b6c  6d6e6f70  | abcdefghijklmnop
0x00599860 :  71727374  75767778  797a6162  63646566  | qrstuvwxyzabcdef
0x00599870 :  6768696a  6b6c6d6e  6f707172  73747576  | ghijklmnopqrstuv
0x00599880 :  7778797a  61626364  65666768  696a6b6c  | wxyzabcdefghijkl
0x00599890 :  6d6e6f70  71727374  75767778  797a6162  | mnopqrstuvwxyzab
0x005998a0 :  63646566  6768696a  6b6c6d6e  6f707172  | cdefghijklmnopqr
0x005998b0 :  73747576  7778797a  61626364  65666768  | stuvwxyzabcdefgh
0x005998c0 :  696a6b6c  6d6e6f70  71727374  75767778  | ijklmnopqrstuvwx
0x005998d0 :  797a6162  63646566  6768696a  6b6c6d6e  | yzabcdefghijklmn
0x005998e0 :  6f707172  73747576  7778797a  61626364  | opqrstuvwxyzabcd
Type c for more data
Any other key will quit this display
```

The user continues to look at the message blocks in the list. The list seems to go on indefinitely. It seems as if *private->last_mp* is being updated correctly, but that *private->first_mp* is not. Looking at *sp_put*, the user sees that *first_mp* is not updated unless *last_mp* is 0 when the list is empty. It seems as if *private->last_mp* was not set to 0 correctly. The user looks at *sp_timeout()* where messages are removed from the list. Indeed, *sp_timeout()* updates only *first_mp*. *last_mp* is not set to zero when the list is empty.

The user changes *sp_timeout()* to check if the list is empty, and sets *private->last_mp* to 0 if it is. The corrected function is shown below.

```
static sp_timeout(private)
struct sp *private;
{
  mblk_t *temp;
  unsigned int s;

  /* Make sure driver isn't being closed. */
  if ((private->sp_state & SPOPEN) && (private->first_mp)) {
      /* Take message off head of queue in private data structure. */
      temp = private->first_mp;
      private->first_mp = private->first_mp->b_next;
      /* The following statement fixes the bug. */
      if (private->first_mp == NULL) private->last_mp = NULL;
      temp->b_next = NULL;
     /* Call putq to put message on sp's read queue and send it upstream.
*/
      putq(private->sp_rdq, temp);
  }
}
```

Example 3: Simple Application Programming Error

In this example, the user writes a test program for the stream described in Example 1. The test program opens several of these STREAMS/UX and *execs* two processes, one that loops doing *putmsgs()* and another that loops doing *getmsgs()*. The test prints a message to the terminal each time it successfully receives 100 STREAMS/UX messages. Some code fragments are shown below.

Put Process

```
/* Initialize the stream and poll structures */
for (i=0; i<stream_count; i++) {
    upper_fd[i].fd = i + OPEN_FILES;
    upper_fd[i].events = POLLOUT;
              .
              .
              .
    }


/* Loop polling to see which STREAMS are writable and writing to them */
while (1) {

    if (poll(upper_fd, stream_count, -1) <= 0) {
       err_handler("Poll returned error %d.\n",errno);
    }

    for (i=0; i < stream_count; i++) {
       if (upper_fd[i].revents) {
          do_a_put(&(str_ctl[i]), &(upper_fd[i]));
       } /* if */
    } /* for */
```

```
} /* while */
```

Get Process

```
/* Initialize the stream and poll structures */
for (i=0; i<stream_count; i++) {
    upper_fd[i].fd = i + OPEN_FILES;
    upper_fd[i].revents = POLLIN|POLLRDBAND;
              .
              .
              .
    }

/* Loop polling to see which STREAMS are readable and reading from them
*/
while (1) {

    if (poll(upper_fd, stream_count, -1) <= 0) {
       err_handler("Poll returned error %d.\n",errno);
    }

    for (i=0; i < stream_count; i++) {
       if (upper_fd[i].revents) {
          do_a_get(&(str_ctl[i]), &(upper_fd[i]));
       } /* if */
    } /* for */

} /* while */
```

The user runs the test, but it does not print any messages. The user runs *strdb* to find the problem.

```
strdb


                          No current structure                S:0
```

The user types *:S* to enter STREAMS/UX subsystem mode.

```
:S

STREAMS subsystem help commands..
?                 - show this help menu
d                 - print status of STREAMS daemon
h                 - show this help menu
la  'name'        - list all active STREAMS on device 'name'
ll  'name' 'minor' - list all drivers linked under the STREAMS
                     driver 'name' and minor number 'minor'
lm  'name' 'minor' - list all modules pushed on STREAMS device
                     'name' and whose minor number is 'minor'
lp  'name' 'minor' - list all drivers persistently linked under
                    the STREAMS device 'name' and minor number
                    'minor'
q                 - quit the STREAMS subsystem commands
qc 'driver' 'file' - print 'driver' read / write side qcount to
                    file
qh  'name' 'minor' - display STREAM head queue structure
                     for device 'name' and minor number 'minor'
```

```
s   [m | d]       - Option d lists all the STREAMS drivers
                     configured in the system. Option m lists
                     all the modules configured in the system
v                 - print version of STREAMS structures
                    displayed
```

Then the user enters the *la* command for *lo* to see what minor number the driver assigned to the stream.

```
la  lo

                          stack empty                          S:0

lo MAJOR = 75
ACTIVE Minor 0x000000  Stream head RQ = 0x005c1500


-- Hit any key to continue --
```

Next the user enters *qh* for *lo* and minor number 0 to start examining the stream. *strdb* formats the stream head read queue.

```
qh lo 0


struct queue 0x5c1500                                          S:1

q_qinfo    =  0x2944f0   q_pad1[2] = 00
q_first    =  0x5e1480   q_other   =  0x5eed74
q_last     =  0x5e1480
q_next     =  0x0
q_link     =  0x0
q_ptr      =  0x76bf00
q_count    =  769
q_flag     =  0x103d
   QREADR
   QFULL
   QWANTR
   QWANTW
   QUSE
   QSYNCH
q_minpsz   =  0
q_maxpsz   =  -1
q_hiwat    =  0x200
q_lowat    =  0x100
q_bandp    =  0x0
q_nband    =  0
q_pad1[0]  = 00
q_pad1[1]  = 00
```

The user notices that the stream head read queue contains several messages that the test program should be able to read. In fact, the queue is full since *q_count* is greater than *q_hiwat*, and the QFULL flag is set.

The user goes back to the code for the get process to see if *poll()* is being called incorrectly. The user checks the parameters passed to *poll()*. The user sees that the initialization code set *revents* instead of *events* before calling *poll()*. *poll()* returns 0 in the *revents* field because no *events* were requested. The corrected code fragment is shown below.

Get Process

```
/* Initialize the stream and poll structures */
for (i=0; i<stream_count; i++) {
    upper_fd[i].fd = i + OPEN_FILES;
   upper_fd[i].events = POLLIN|POLLRDBAND; /* Changed revents to events
*/
                .
                .
                .
    }
```

HP-UX Kernel Debugging Tools

This section describes the HP-UX kernel debugging tools and techniques available for HP-UX release 10.0. These tools and techniques may change from release to release. This manual will focus primarily on the Series 700 and 800 debugging tools and techniques. Sources of additional information for the Series 700 are cited below.

Kernel level debugging is associated with the hardware that a kernel is running on. The kernel level debugging tools are different for the different hardware platforms.

For the Series 700, kernel level debugging may be performed using *ddb*. *ddb* can be used to set breakpoints, single-step through code, examine the contents of data structures at key points, change the contents of structures and variables, and use most other normal debugging techniques. *ddb* is documented in *HP-UX Driver Development Guide*, part number 98577-90013. *ddb* is not part of the standard Series 700 HP-UX product. To obtain a copy of *ddb* software, contact your HP representative.

For Series 700 and 800, kernel level debugging may be performed using *adb*, which is a general purpose assembly language debugging program. *adb* allows you to look at HP-UX files and system core files that result from system panics, to examine system registers and memory locations as they were at the time of the panic, and to print data from these files in a variety of formats. *adb* can also be used to examine a running HP-UX system. *adb* is part of the standard HP-UX product and is located in */usr/bin* on every HP-UX system. It is important to use the revision of *adb* which corresponds with the release of the kernel being debugged -- for example, a 9.0 version of *adb* will not work well on a 10.0 kernel. This chapter describes in detail how to use *adb* to debug kernel problems on Series 700 and 800 systems. For additional information on *adb*, refer to the following items:

- adb(1) man page
- *ADB Tutorial*, part number 92432-90005
- *Assembly Language Reference Manual*, part number 92432-90001
- *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, part number 09740-90039

- *PA-RISC Procedure Calling Conventions Reference Manual*, part number 09740-90015

HP-UX Kernel Debugging Tools and strdb

The *strdb* tool can be used in conjunction with other standard HP-UX kernel debugging tools to provide STREAMS/UX-specific information and data formatting. Generally, if your system is running normally except for STREAMS/UX, it is recommended that you use *strdb* to debug the problem. If your system panics or hangs, *strdb* can be used on the resulting system core dump, along with *adb* to diagnose the problem. *strdb* is documented earlier in this chapter, and examples of using *adb* and *strdb* together are given at the end of this chapter.

What Is a System Panic?

Unlike user code, programming errors in kernel code can cause system panics. A system panic will result in a panic message to the console. Also, a system core dump will be generated. This is a copy of physical memory at the time of the panic. The panic message and core dump can be examined using *adb* and *strdb* to determine the cause of the panic.

There are three main categories of panics. The first category is when a kernel routine calls *panic()* because of a system inconsistency from which it cannot recover. In this case, the panic message contains a string from the routine that called *panic()*, explaining why panic was called. In the example below, the panic string is “ifree: freeing free inode.” A hexadecimal stack trace will also be printed. Interpreting the stack trace will be described later.

```
System Panic:
@(#)9245XA HP-UX (A.10.00) #1: Wed Sep 28 15:47:13 PDT 1994
panic: (display==0xb000, flags==0x0) ifree: freeing free inode

PC-Offset Stack Trace (read across, most recent is 1st):
  0x0014766c  0x001480b0  0x000b3a38  0x000b411c  0x000b3b78  0x000b76
5c
  0x000b10d8  0x000aefd0 0x0001c500
End Of Stack
```

The second category is the occurrence of a kernel level trap or exception condition. These usually involve virtual memory and are described below. A hexadecimal stack trace is also printed.

The third is the occurrence of a High Priority Machine Check (HPMC), which usually indicates a hardware problem. An HPMC is characterized by a total, sudden system halt and an HPMC “tombstone” printed on the console, which records the contents of the system's registers. If you encounter an HPMC, contact your HP service representative. Note that an HPMC tombstone is also printed out after a TOC (Transfer of Control -- see “Transfer of Control In Case of System Hang” for details). There is no need to contact an HP representative for an HPMC tombstone that is the result of a TOC.

Traps

Some very common panics occur from either the trap routing or interrupt routing routines. Whenever this low level code detects a trap occurring in the system and it believes that it cannot be corrected, it will panic the machine. The most common faults are described below.

Data Segmentation Faults

Usually, a data segmentation fault occurs when a process (in kernel mode) attempts to dereference a null pointer. If you receive a data segmentation fault, information similar to the following will be printed on the system console:

```
trap type 15, pcsq.pcoq = 0.85b7c, isr.ior = 0.4
@(#)9245XA HP-UX (A.10.00) #0: Sat Aug 13 23:17:54 PDT 1994
panic: (display==0xbf00, flags==0x0) Data segmentation fault
```

pcsq.pcoq is the current instruction address, and *isr.ior* is the current data address. This trap message means that the instruction at location 0x85b7c tried to reference address 4 in space 0. You could look in *adb* to see what the instruction was trying to do. The instruction may have been attempting to get a value 4 bytes off of some pointer. Because of a possible logic problem, the pointer might not have been initialized.

Instruction Page Faults

An instruction page fault occurs when a process in kernel mode jumps to an address which is not mapped, and tries to execute it. Because the page is not mapped, and the kernel is not paged, a fault is generated. This would appear as the following:

```
trap type 6 pcsq.pcoq = 0.0 isr.ior = 4.78
@(#)9245XA HP-UX (A.10.00) #0: Sat Aug 13 23:17:54 PDT 1994
panic: (display==0xbf00, flags==0x0) Instruction page fault
```

The *pcsq.pcoq* pair is important; the user attempted to jump to page zero and start executing. In this case, because the fault was an instruction page fault, the *isr.ior* pair is meaningless. The page fault may have occurred because of an indirect procedure call, where the address of the routine to be called was not initialized.

Protection Violations

A third common panic is the protection violation. This type of panic occurs when the kernel tries to reference a data structure that does not belong to the current process. This panic also occurs if the kernel attempts to reference an object in a way which is not permitted by the access rights assigned to the page where the object resides, for example, an attempt to write on a read-only page. Another frequently overlooked area of protection faults are unaligned access violations. These appear to be protection faults, but are caused by performing an operation on an unaligned address, for example, load word on a non-word aligned address. In each of these cases, trap type 18 or 7 would be generated. The *pcsq.pcoq* pair would give the offending instruction, and the *isr.ior* would give the offending data address referenced.

Generating and Retrieving System Core Dumps

HP-UX will attempt to create a snapshot of physical memory and register contents before it stops running. This snapshot can assist engineers in determining the cause of the problem because it holds a record of what the system was doing at the time it crashed. The correct name for this snapshot is a *core dump*. The default location for this snapshot is the primary swap area, but it is possible to configure systems to put the snapshot on another disk device. See the *System Administration Tasks* manuals for the Series 700 and the Series 800 for information on configuring dump devices.

A core dump is composed of two files, a core file and an object file. The core file is an image of the system's physical memory and register contents at the time of a crash. The object file is the kernel file, */stand/vmunix*.

To retrieve a core dump, the program */usr/sbin/savecore* must be executed. *savecore* will retrieve the core file from the swap device, along with a copy of the system's kernel file, and save both in a specified directory. The core file and the kernel file make up the core dump pair (for example, *vmcore.N* and *vmunix.N* where *N* is a number that associates a core dump pair).

adb and *strdb* require that both members of a core dump pair be present. In addition, it is very important that these members match for *adb* and *strdb* to be effective. They must match because the kernel (*vmunix.N*) file contains information which is used by *adb* and *strdb* as a road map into the core (*vmcore.N*) file.

Setting Up Your System To Save a Core Dump

In order to have system core dumps saved automatically during boot-up, the *savecore* function must be enabled in the system's */etc/rc.config* file. Search this file for the string SAVECORE and follow the instructions in the comments.

Manually Getting a Core File from the Swap Partition

If *savecore()* was not run at boot-up, or did not succeed, you can still run savecore(1m) manually by taking the following steps:

```
/usr/bin/bdf                    # find enough space for the dump
mkdir /tmp/syscore              # assuming /tmp has enough space
/usr/sbin/savecore /tmp/syscore # savecore to the chosen directory
```

savecore begins by reporting the date and time of the crash. Next, it looks in the specified directory for a file named *bounds*. The *bounds* file contains the next sequence number (N), which *savecore* will use to create a unique core file and kernel file. *savecore* will copy the core image from the primary swap device to a file named *vmcore.N*. Lastly, *savecore* copies */stand/vmunix* to a file named *vmunix.N* to complete the core dump pair.

Problems Encountered In Saving/Obtaining a Core Dump

If */stand/vmunix* was not the kernel that was running when the crash occurred, savecore will exit quickly without printing any message and will not save the core file. Use the -d option to tell *savecore* what kernel was running at the time of the system crash:

```
/usr/sbin/savecore -d /stand/vmunix.bad /var/adm/crash
```

If a core dump pair is incomplete or not saved after a panic, you can look to the savecore(1m) man page for help.

Transfer of Control In Case of System Hang

A system hang is a situation in which the system seems to be up but does not respond to external user control. Should this happen to your system, you will want to obtain a core dump so that the cause for the hang can be analyzed. The method for obtaining a core dump of a kernel in this state is to use the Transfer of Control (TOC) mechanism. The TOC mechanism causes the machine to vector through a special address which will cause the machine to do a core dump. Most Series 700 and 800 machines have the capability to perform TOC but the methods for performing this task are machine-dependent:

- **Series 800 Models 850, 855, 86x, F, G, H, I, and 870**: If you have an access port connected to your machine, you must enable it through your front panel. Next,

type a “Control b” on the console. This will put your console under the supervision of the access port. You will get a “CM>” prompt, at which you may type “TC.”

- **Series 800 Models 834/5, 845, and 8x2** have a key-operated TOC mechanism. To execute a TOC, turn the key all the way to the right (clockwise).
- **Series 800 Models 808 and 815** have a button-operated TOC mechanism. From the rear of the machine, look for this button on the lower right-hand side (it will be marked TOC). You will need an object, like a pen, to push the TOC button.
- **Series 700s** have a button-operated TOC mechanism. The button is on the right side front of the computer. Pull open the door covering the system activity LED's, and the TOC button is the small white button on the far left.

Core File Size Requirements

It is best if the size of the *vmcore.N* file is equal to that of the machine's physical memory. Because the core file is an image of memory at the time of a crash, if its size is not equal to the machine's physical memory size, some information will be lost. However, it is still possible to get some information from a partial core dump.

Symbol Information

Make sure that the *vmunix.N* file has not been stripped. *adb* and *strdb* will not work without symbol information. Use the *file* command to confirm that the symbols have not been stripped:

```
file vmunix.0
vmunix.0        s800 executable        -not stripped
```

Using adb

This section describes how to use *adb* on core dumps obtained following a system crash. See “Generating and Retrieving System Core Dumps” for information on how these dumps are obtained. *adb* can also be used to examine a system that is currently running.

See the adb(1) man page or *ADB Tutorial* for more information.

Invoking adb

When using *adb* on a system core dump, you must use the “-k” option. This option will tell adb to treat the core dump as a system core dump instead of a user process core dump, which is organized differently. For example, to call *adb* on the dump pair *vmcore.1* and *vmunix.1*, perform the following:

```
adb -k vmunix.1 vmcore.1
```

When using *adb* on a running HP-UX system, you also use the “-k” option, and use */stand/vmunix* as the object file and */dev/mem* as the core file:

```
adb -k /stand/vmunix /dev/mem
```

You will probably need to be superuser to access */dev/mem*. Because you are examining a running (and continuously changing) system, *adb* will not be able to set you up in any specific process context, but you will be able to examine kernel global variables.

Context on Entry to adb

adb maintains a set of registers corresponding to the registers of the machine. The *adb* command *$r* will print out the values of these registers. When *adb* is invoked on a system core with the *-k* option, it sets these registers to the values of the machine registers at the time the system core dump was taken. These register values are not the values the registers contained at the point the panic or trap occurred. Instead, they are the values the registers contained at the time the kernel started dumping a copy of physical memory to the swap area. How to use these “dump time” register values to determine the state of the registers at the time the trap or panic

occurred will be described later. These “panic time” register values enable the user to examine the context of the process that was running at the time of the system crash.

Debugging Hung Systems

If the system core dump is from a transfer of control (TOC) of a hung system, adb will be unable to determine the “dump time” or “panic time” register values. In these cases, *adb* can still be used to determine the contents of the kernel message buffer (see “Finding the Panic Message”), and to examine kernel global variables (see “Obtaining Important Kernel Global Variables”), but it will not be able to give you a stack trace or context for the process that was running at the time of the system crash.

It is especially important, when looking at a dump from a system which appeared to be hung, to check the kernel globals *freemem*, *freemem_cnt*, and *avenrun*. These variables may indicate that your system was out of memory or was overloaded. (See “Obtaining Important Kernel Global Variables” for more information.)

It can also be helpful, before doing a TOC on a system which appears hung, to determine how complete the system paralysis is. The following table describes hang symptoms, from the least severe to the most severe. This table may help you determine where your system fits on this continuum.

Symptom	Explanation
Some processes, like your shell or your tests, do not run, but other processes are running.	Your system is not hung, but there is some other problem holding back your processes. If you have a terminal session that is working, use *strdb* and *adb* to look at the kernel and the STREAMS/UX subsystem state.
You cannot login, either locally or remotely.	Your system may not be hung, its networking software state, terminal I/O or *getty* processes may be deadlocked in some way. If you have a terminal session that is working, use *strdb* and *adb* to look at the kernel and the STREAMS/UX subsystem state.

Symptom	Explanation
You cannot ping your system.	Your system may not be hung, its networking software state may be deadlocked in some way. If you have a terminal session that is working, use *strdb* and *adb* to look at the kernel and the STREAMS/UX subsystem state.
Carriage returns do not echo on the console or on other login sessions.	Your system is hung, but is probably TOC-able. TOC the system and examine the kernel globals in the dump.
Your system has an LED activity display which is not being updated; it is showing no system activity at all.	Your system is hung, but is probably TOC-able. TOC the system and examine the kernel globals in the dump.
Your system has an access port enabled, and typing *CTRL-b* on the console gives no response, or you attempt to TOC a system without an access port with no success.	Your system is ignoring very high-level interrupts, and it is so thoroughly hung that you will probably be unable to TOC it. Hangs as severe as this are extremely rare. Hit the system reset button, and try to debug the problem using other methods such as code reviews, *panic*s, or *printf*s.

Finding the Panic Message

The kernel maintains a circular message buffer into which text can be printed using the kernel *printf*, *msg_printf*, and *cmn_err* routines. At the time of a panic, a panic message is printed to this buffer. A stack trace consisting of instruction addresses in hexadecimal is also printed out, as well as the current instruction and data addresses being accessed at the time of the crash. Other interesting information may also be located in the buffer, such as system boot-up messages and kernel error messages that may help pin down the cause of the panic. To print out this buffer, invoke *adb* on the system dump and type the following:

```
msgbuf+10/s
```

Examples of *msgbuf* contents are included in the examples at the end of this chapter.

Interpreting the Panic Stack Trace

adb can be used to translate the hexadecimal stack trace printed after the panic message into procedure addresses. For each hexadecimal number in the stack trace, use the *adb i* command to determine where in the kernel the address occurs. For example, the hex stack trace below can be deciphered as follows:

```
PC-Offset Stack Trace (read across, most recent is 1st):
 0x0016da70 0x000e5a68 0x000d34cc 0x0009ea14 0x00099714 0x0009
2fdc
 0x0006e0c8 0x0006dbb8 0x0006d2a8 0x001954e8 0x00194fa4 0x000b
7e24
 0x001846d4 0x00181730 0x00156538 0x00156af8 0x001567b8 0x000e
6d80
 0x000d3aac
 End Of Stack
```

In *adb* (text preceded by "#" are comments):

```
0x0016da70/i                              # use of adb i command
panic+30:       addil   -1000,dp          # adb's response
0x000e5a68/i
trap+0xADC:     b       trap+1004
0x000d34cc/i
$call_trap+20:  rsm     1,r0
0x0009ea14/i
flushq+60:      ldbs    0xD(r21),r22
0x00099714/i
q_free+1C:      ldw     -0xA4(sp),r31
```

Manual Stack Back-Tracing

You may need to use *adb* to manually back-trace your stack. This is necessary when the hexadecimal stack trace printed by *panic* is incomplete. For example, *panic* may print a few hex addresses and then the message:

```
stktrc:  cannot find descriptor
```

or

```
stktrc: cannot find rp
```

You may also need to do a manual stack back-tracing if you wish to find out how the arguments the routines in your stack trace were called. You will need the value of the stack pointer for each routine in the stack and manual stack back-tracing will tell you these values.

PA-RISC Procedure Calling Conventions Overview

The following is a very brief overview of the PA-RISC procedure calling convention. More information can be obtained from the *PA-RISC Procedure Calling Conventions Reference Manual*.

PA-RISC machines have 32 general use registers. These registers are identical physically, but are assigned different roles by the PA-RISC operating systems and compilers in order to enable procedure calls to take place efficiently and consistently. The following table lists these special roles:

Table 5 General Use Register Roles

r0	Value is always zero.
r1	Scratch register.
r2	Return pointer, also known as *rp*. This is the instruction address the called procedure will return to when it is finished executing.
r3 - r18	Callee saves. If the called procedure wishes to modify any of these registers, it must save the original contents on its stack and restore the contents before returning to the caller.
r19 - r22	Caller saves. The called procedure is free to modify these registers without saving the original contents. If the calling procedure wants to retain the contents, it must save them before making the procedure call and restore them after the call returns.
r23 - r26	First four procedure arguments, also known as *arg0*, *arg1*, *arg2*, and *arg3*. The calling procedure loads the first four procedure arguments into these registers before making the procedure call.
r27	Global data pointer, also known as *dp*.
r28 - r29	Procedure return values, also known as *ret0* and *ret1*. The called procedure loads the return values into these registers before returning.
r30	Stack pointer, also known as *sp*.
r31	Millicode return pointer, or scratch register.

The only registers you need to be concerned with for manual stack back-tracing are r2 (*rp*) and r30 (*sp*), although the other registers become important when trying to determine what arguments a procedure in the trace was called with.

In order to implement these register roles, at the start of each procedure a stack frame is allocated and *callee save* registers which the called procedure is planning to modify are stored in the stack frame. The stack frame is allocated simply by incrementing the *sp* by the size of the stack frame needed, using either the *stwm* or *ldo* instruction. For example, below are the instructions which create the stack frame for *ioctl*. Numbers in brackets ([ ]) refer to the notes below.

```
ioctl:          stw     rp,-14(sp)    [1]
ioctl+4:        stwm    r3,100(sp)    [2]
ioctl+8:        stw     r4,-0xFC(sp)  [3]
ioctl+0xC:      stw     r5,-0xF8(sp)  [4]
ioctl+10:       stw     r6,-0xF4(sp)  [5]
```

[1] Store return instruction address at 0x14 above the caller's stack pointer. Note that the return address is stored in the caller's stack frame, not the callee's stack frame.

[2] Store the contents of r3 at the current *sp*, then allocate the stack frame by adding 0x100 to sp. The *stwm* instruction stands for store word and modify.

[3] Store the contents of r4 at *sp* - 0xFC, just below where you stored r3.

[4] Store the contents of r5 at *sp* - 0xF8, just below where you stored r4.

[5] Store the contents of r6 at *sp* - 0xF4, just below where you stored r5.

The instruction *ldo* (load offset) can be used instead of *stwm* for allocating the stack. For example:

```
doadump:        stw     rp,-14(sp)  [1]
doadump+4:      ldo     30(sp),sp   [2]
```

[1] Store return instruction address in caller's stack frame.

[2] Add 0x30 to the current value in register *sp* and store the result in *sp*, allocating stack frame.

Basic Stack Back-Tracing

Given the stack pointer, *sp*, and the current instruction address, *pcoqh*, it is possible to get the previous stack pointer and instruction address. The starting values for *sp* and *pcoqh* are obtained from the *adb $r* command. As mentioned above, when *adb* is invoked on a system core with the *-k* option, it sets these registers to the values of the machine registers at the time the system core dump was taken. The *$r* command prints out these registers. Below are the first few lines of the *$r* display.

```
pcsqh 0         pcoqh     24B34 doadump+0xEC
pcsqt 0         pcoqt     0     _fp_status
rp    0xDBF48  panic_boot+354

arg0  1         arg1   0xC57B             arg2   2000     arg3
9BD70152
sp    20F380    ret0 303847        ret1  797      dp    1F6000
```

There are four steps to back-tracing a stack:

1 Determine the size of the current stack frame.

The size of the current stack frame is simply the amount the *sp* is incremented at the entry to the current procedure. To find that number, use *adb* to print out the first few instructions of the current procedure. To determine the initial current procedure, look at the value of the register *pcoqh*, which appears at the end of the first line of the $r output. In most cases, this initial procedure will be doadump.

```
doadump/3i
doadump+3:      stw     rp,-14(sp)
                ldo     30(sp),sp
                mfctl   iva,r22
```

doadump's second instruction is an *ldo* which increments the stack pointer by 0x30, so doadump's stack frame size is 0x30.

2 Determine the previous stack pointer.

The previous stack pointer is the current stack pointer, minus the current stack frame size. *adb* can be used to keep track of the *sp* register by calculating the previous stack pointer using the following adb commands:

```
<sp-0x30>sp     [1]
.=X             [2]
        20F350  [3]
```

[1] Take the current value of the *sp* register, decrement it by 0x30, and store the result back into the *sp* register. See *adb* documentation for more information on adb registers and the “<“ and “>” operators.

[2] Print out the new value of *sp*. This information should be saved in case you need to find out the contents of registers which have been pushed onto the stack frame. See *adb* documentation for more information about the concept of ".", the current location in the core file.

[3] *adb* output in response to the previous command, *.=X*

3 Find the current return pointer.

Your current procedure is doadump, and you have just set *sp* so that it is the same value it was when doadump was first entered, before the *ldo* instruction was executed. Recall that doadump's first instruction is:

```
stw     rp,-14(sp)
```

Because you have just set *sp* to the same value it had when doadump's first instruction was executed, you can find the *rp* by looking at what is in *sp*-0x14:

```
<sp-0x14/X                                      [1]
crash_monarch_stack+1EC:        0xDBF48         [2]
```

[1] Print out the value of the location *sp*-0x14 in hexadecimal.

[2] *adb*'s response. *crash_monarch_stack+1EC* can safely be ignored. 0xDBF48 is the instruction address which was in *rp*.

4 Find out which procedure the return pointer points to.

The *adb i* command will tell you this:

```
0xDBF48/i                                                       [1]
panic_boot+354: comibt,=,n      0,ret0,panic_boot+368           [2]
```

[1] use of the *i* command

[2] *adb*'s response

Notice that the *$r* command has already indicated that *rp* corresponds to *panic_boot+354*.

To continue back-tracing the stack, iterate the four steps shown above. Here is the *adb* sequence of commands and responses to trace the next two levels back in this stack. Text preceded by "#" are comments.

```
panic_boot/3i                       # look at beginning of
panic_boot:                         #  panic_boot for stack frame
panic_boot:   stw    rp,-14(sp)     #  size
              stwm   r3,80(sp)      # stack frame size is 0x80
              stw    r4,-7C(sp)
<sp-0x80>sp                         # calculate new sp
.=X                                 # print out new sp
              20F2D0
<sp-0x14/X                          # find rp in caller's
```

```
crash_monarch_stack+16C: 0xDB938 #    stack frame
0xDB938/i                        # what instruction address
boot+24:     addil   0,dp        #    does rp correspond to?
boot/3i                          # look at beginning of boot
boot:                            #    for stack frame size
boot:        stw    rp,-14(sp)
             stwm   r3,80(sp)    # stack frame size is 0x80
             stw    r4,-7C(sp)
<sp-0x80>sp                      # calculate new sp
.=X                              # print out new sp
             20F250
<sp-0x14/X                       # find rp in caller's
crash_monarch_stack+0xEC: 1518A4 #    stack frame
1518A4/i                         # what instruction address
panic+0xF0:  ldw    -94(sp),rp   #    does rp correspond to?
panic/3i                         # look at beginning of panic
panic:                           #    for stack frame size
panic:       stw    rp,-14(sp)
             stwm   r3,80(sp)    # stack frame size is 0x80
             stw    r4,-7C(sp)
```

If you are doing a manual stack back-trace in order to find out values of registers which have been pushed onto the stack, it is useful to save the results of the four steps at each iteration for future reference. A table such as the following can be helpful:

sp	pcoqh	Procedure Address	Frame Size
20F380	24B34	doadump+0xEC	0x30
20F350	0xDBF48	panic_boot+354	0x80
20F2D0	0xDB938	boot+24	0x80
20F250	1518A4	panic+0xF0	0x80

Exceptions to the Four Steps

The four basic steps of stack back-tracing have some exceptions:

- **panic**: If your procedure address is in panic, you need to take special steps to find out the true value of your current stack pointer. Instead of being the previous *sp* minus the previous frame size, panic's *sp* can be found at location *panic_save_state*. Do the following to find the value using adb and reset adb's copy of sp:

  ```
  panic_save_state/X                        [1]
  panic_save_state:                         [2]
  panic_save_state:              7FFE6F48
  7FFE6F48>sp                               [3]
  ```

  [1] Ask *adb* to print out location *panic_save_state* in hex.

[2] These two lines are *adb*'s response. panic's actual *sp* is 7FFE6F48.

[3] Reset *sp* to the correct address.

Now that you have panic's real stack pointer, the other steps in the back-tracing process can be executed normally. Text preceded by "#" are comments.

```
<sp-0x80>sp                                 # calculate new sp
.=X                                         # print out new sp
                    7FFE6EC8
<sp-0x14/X                                  # find rp in caller's
7FFE6EB4:           0xDF108                 #    stack frame
0xDF108/i                                   # what instruction address
trap+0xA28:         b      trap+0xF18       #    does rp correspond to?
trap/3i                                     # Look at beginning of trap
trap:                                       #    for stack frame size
trap:               stw    rp,-14(sp)
                    stwm   r3,100(sp)       # stack frame size is 0x100
                    stw    r4,-0xFC(sp)
<sp-0x100>sp                                # calculate new sp
.=X                                         # print out new sp
                    7FFE6DC8
<sp-0x14/X                                  # find rp in caller's
7FFE6DB4:           0xD0BD4                 #    stack frame
0xD0BD4/i                                   # what instruction address
$call_trap+20:      rsm     1,r0            #    does rp correspond to?
```

- **$call_trap, $call_int, $ihndlr_rtn, $thndlr_rtn, $RDB_trap_patch, $RDB_int_patch**: These procedures do not follow the ordinary procedure calling conventions. They are written in assembly language, and are used to create a *save state* structure which saves the values of all registers at the time of a trap or an interrupt. The *save state* is then passed to *trap()* or the appropriate interrupt routine. The *save state* starts at *sp* - 0x230, and you can retrieve the previous stack pointer and current *pcogh* from the *save state*, as shown below. The offsets into the *save state* are for the 10.0 release, and may change from release to release.

```
<sp-0x230>sp                                [1]
<sp+0x84/X                                  [2]
7FFE6C1C:       96B70                       [3]
<sp+0x78/X                                  [4]
7FFE6C10:       7FFE6B98                    [5]
7FFE6B98>sp                                 [6]
96B70/i                                     [7]
qenable+10:     ldws    0(r20),r21
qenable/3i
qenable:
qenable:        stw     rp,-14(sp)
                ldo     80(sp),sp
                stw     arg0,-0xA4(sp)
```

[1] Reset *sp* to point to the top of the save state structure.

[2] Save state structure + 0x84 is the location of the *pcogh*.

[3] *adb*'s response -- 96B70 is the return instruction address.

[4] Save state structure + 0x78 is the location of the *sp*.

[5] *adb*'s response -- 7FFE6B98 is the current stack pointer.

[6] Reset *sp* to the correct value.

[7] Continue to iterate the four basic stack back-tracing steps.

The table of results from the back-tracing so far should look like this:

sp	pcoqh	Procedure Address	Frame Size
20F380	24B34	doadump+0xEC	0x30
20F350	0xDBF48	panic_boot+354	0x80
20F2D0	0xDB938	boot+24	0x80
7FFE6F48	1518A4	panic+0xF0	0x80
7FFE6EC8	0xDF108	trap+0xA28	0x100
7FFE6DC8	0xD0BD4	$call_trap+20	
7FFE6B98	96B70	qenable+10	0x80

Mapping Assembly Language Locations to Source Code Lines

Once you know the instruction address location where the system panic or trap occurred, the troubleshooting step is to find where in the source code the panic or trap occurred. For panics, search the source code for the panic which uses the same string that was printed out when the kernel panicked. This will tell you exactly where the panic occurred in the source code. The method for traps is to use *adb* to print out the procedure in which the trap occurred in assembly language. Then, work backwards from the instruction address, looking for clues in the assembly instructions which will help pinpoint the corresponding location in the source. The most useful clue is a branch to another procedure. In PA-RISC, branches are done with the branch and link instruction, bl, and in assembly a branch will look like this:

```
bl      copen,rp   [1]
```

[1] a procedure call to *copen()*

or:

```
bl      creat+34,rp  (save_pn_info)  [1]
```

[1] a procedure call to *save_pn_info()*

By comparing the branches in the assembly code before and after the instruction where the trap occurred with the procedure calls in the source code, the corresponding source code line can often be determined. See the examples at the end of this chapter for more details.

Other useful assembly code landmarks are the use of the *extru*, *extrs*, *zdep*, and *ldws* instructions in checking and setting flag bits, and the use of the compare and branch instructions, *comb*, *combf*, *combt*, *comib*, *comibf*, and *comibt*, to implement if statements. For example, the *ioctl()* source code:

```
if ((fp->f_flag & (FREAD|FWRITE)) == 0)
```

is implemented by the assembly code:

```
ioctl+60:       ldws    0(r8),r13               [1]
ioctl+64:       extru   r13,1F,2,r14            [2]
ioctl+68:       comibf,=,n      0,r14,ioctl+80  [3]
```

[1] Load from memory address pointed to by r8, into r13.

[2] Extract 2 bits from r13, starting at bit 1F, place bits in r14.

[3] If r14 is not zero, branch to *ioctl*+0x80.

In the example above, *fp* is in r8. If *fp* were null, a trap type 15 would occur at *ioctl*+60, when attempting to load off of a null pointer.

For more information about PA-RISC assembly language, see the *Assembly Language Reference Manual* (part number 92432-90001), the *PA-RISC 1.1 Architecture and Instruction Set Reference Manual* (part number 09740-90039), or the *PA-RISC Procedure Calling Conventions Reference Manual* (part number 09740-90015).

Obtaining Procedure Argument Values

It is often useful in debugging a problem to know what parameter values a procedure in the stack trace was called with. For example, in the following stack trace it would be useful to know the arguments *flushq()* was called with.

```
panic+30:       addil   -1000,dp
trap+0xADC:     b       trap+1004
$call_trap+20:  rsm     1,r0
flushq+60:      ldbs    0xD(r21),r22
q_free+1C:      ldw     -0xA4(sp),r31
```

Obtaining the First Four Arguments

Arguments 0 through 3 are passed from the calling procedure to the called procedure by loading the values into registers 23 - 26. These registers are also known as *arg0*, *arg1*, *arg2*, and *arg3*. For example, here is *bmap()* preparing to call *realloccg()* by moving *realloccg()*'s arguments from the registers they are in to the argument registers by doing an *or* on the source registers with r0, which is always zero:

```
bmap+16C:       or      r10,r0,arg1
bmap+170:       or      ret0,r0,arg2
bmap+174:       or      r8,r0,arg3
bmap+178:       or      r4,r0,arg0
bmap+17C:
```

Next, here is *flushq()* preparing to call *rmvq()* by loading *arg0* and *arg1* from its stack frame. Note that *arg1* gets loaded in the delay slot of the branch instruction *bl*. See the *Assembly Language Reference Manual* or the *PA-RISC 1.1 Architecture and Instruction Set Reference Manual* for more information on branch delay slots.

```
flushq+0xE0:    ldw     -64(sp),arg0
flushq+0xE4:    bl      rmvq,rp
flushq+0xE8:    ldw     -34(sp),arg1
```

After allocating its stack frame and saving any callee save registers, the called procedure will usually load the argument registers into some of the callee save registers that it just saved the values of. For example, here is *realloccg()* saving the contents of the *callee save* registers r3 - r10 and loading *arg0* - *arg3* into some *callee save* registers.

```
realloccg:      stw     rp,-14(sp)
realloccg+4:    stwm    r3,80(sp)
realloccg+8:    stw     r4,-7C(sp)
realloccg+0xC:  stw     r5,-78(sp)
realloccg+10:   stw     r6,-74(sp)
realloccg+14:   stw     r7,-70(sp)
realloccg+18:   stw     r8,-6C(sp)
realloccg+1C:   stw     r9,-68(sp)
realloccg+20:   stw     r10,-64(sp)
realloccg+24:   or      arg0,r0,r3
realloccg+28:   or      arg1,r0,r6
realloccg+2C:   or      arg2,r0,r7
realloccg+30:   or      arg3,r0,r4
```

Here is *rmvq()* storing its arguments away in its stack frame:

```
rmvq:           stw     rp,-14(sp)
rmvq+4:         ldo     80(sp),sp
rmvq+8:         stw     arg0,-0xA4(sp)
rmvq+0xC:       stw     arg1,-0xA8(sp)
```

If the arguments were put into *callee save* registers, the next procedure up in the stack trace will save these registers in its stack frame. You can retrieve these values from the stack. If the arguments are stored on the stack frame, you can also retrieve them from the stack. But first you must make sure that the contents of the *callee save* registers or the stack frame locations you are interested in were not modified between the time the arguments were loaded at the beginning of the procedure and the time the next procedure call on the stack trace took place. The easiest way to determine this is to have *adb* print out the assembly code for the procedure into a file and use an editor such as vi to find all references to the register between the beginning of the procedure and the branch to the next procedure in the stack trace. If none of these references modify the register, the value which the next procedure has saved in its stack frame is valid.

To print the assembly of a procedure to a file using *adb*:

```
$>filename              [1]
procedure,100/ia        [2]
$>                      [3]
```

[1] Tell *adb* to direct *stdout* to the file *filename*. There should be no space between $> and the filename.

[2] Print the first 0x400 instructions of procedure.

[3] Set *stdout* back to the terminal.

Now, edit *filename*, and search for all instances of the register or stack frame location of interest. Any instruction which would modify the contents of the register could potentially overwrite the information you are trying to get. Below are some examples of modifying instructions. Note that in all cases the register being modified, also known as the target register, is the last register in the instruction.

```
ldw     10(r3),r4       will overwrite r4
ldhs    4(r3),rp        will overwrite rp
ldo     -1(r20),r22     will overwrite r22
ldwx    r31(arg3),r21   will overwrite r21
or      r3,r0,arg0      will overwrite arg0
extrs   ret1,1F,10,r21  will overwrite r21
zdep    r20,1A,1B,r31   will overwrite r31
sub     r31,arg1,r31    will overwrite r31
sh3add  arg1,r0,r31     will overwrite r31
stw     r19,-38(sp)     will overwrite memory location sp - 0x38
```

Sometimes an instruction which modifies the register of interest can appear to occur between the beginning of the procedure and the call to the next procedure in the stack because of how the assembly code is laid out. However, the modifying instruction actually would not have been executed because it was part of a conditional code path that was not taken. For example, this C code from *ioctl()*:

```
if ((fp->f_flag & (FREAD|FWRITE)) == 0) {
        u.u_error = EBADF;
        return;
}
```

compiles into this assembly:

```
ioctl+60:       ldws    0(r8),r13
ioctl+64:       extru   r13,1F,2,r14
ioctl+68:       comibf,=,n      0,r14,ioctl+80
ioctl+6C:       ldw     68(r3),r19
ioctl+70:       ldo     9(r0),r21
ioctl+74:       sth     r21,312(r19)
ioctl+78:       b       ioctl+7F0
ioctl+7C:       ldw     -1D4(sp),rp
ioctl+80:       ldws    4(r5),r7
```

If the *if* statement is false, the branch at *ioctl*+68 is taken, and instruction *ioctl*+6C is never executed because the *,n* in *ioctl*+68 causes the instruction in the branch delay slot to be nullified, or not executed. ioctl+70 through ioctl+7c are never executed because the branch at ioctl+68 branches past these instructions to ioctl+80. If ioctl+6c through ioctl+7C had been executed, r19, r21, and rp would have been modified.

Suppose you have determined that the procedure whose arguments you are interested in does not modify the registers it loaded the arguments into before the next procedure call in your stack. You can look at the appropriate location in the stack frame of the next procedure call in the stack to get the value. For example, if a routine whose registers you are interested in has called panic, you look at the beginning of panic's assembly to see which *callee save* registers it saves in its stack.

```
panic:          stw     rp,-14(sp)
panic+4:        stwm    r3,40(sp)
panic+8:        stw     r4,-3C(sp)
panic+0xC:      stw     r5,-38(sp)
panic+10:       stw     r6,-34(sp)
```

Obtain panic's *sp* by manual stack back-tracing, and then r3 is at *sp* - 0x40, r4 at sp - 0x3C, and so on.

Obtaining Arguments 5 through N

Only the first four arguments to a procedure are passed via registers. Any remaining arguments are pushed onto the calling procedure's stack frame, where the called procedure will retrieve them. If you have the calling procedure's *sp* you can use *adb* to get the values of the arguments. For example, *symlink()* calls *lookuppn()*, which has six arguments. Here is the assembly code which sets up the six arguments:

```
symlink+40:     stw     r4,-34(sp)
symlink+44:     stw     r3,-38(sp)
symlink+48:     ldo     -3C(sp),arg2
symlink+4C:     ldo     -9C(sp),arg0
symlink+50:     or      r0,r0,arg1
symlink+54:     bl      rename+34,rp  (lookuppn)
symlink+58:     or      r0,r0,arg3
```

If you want to get the fifth argument, you see that *symlink()* places it in its stack frame at sp - 0x34. Argument 5 is at -0x34 because the procedure calling convention specifies that arguments get placed in the stack frame in reverse order, so arg6 is at sp - 0x38, just above arg5, and if *lookuppn()* had seven arguments, arg7 would be placed at sp - 0x3C. If you know *symlink()*'s sp from doing a manual stack back-trace, you can use it to get the value of argument 5:

```
7FFE6B98-0x34/X
7FFE6B64:       2D7298  # adb's response
```

Obtaining Register Contents from Trap save_state or panic_save_state Areas

If the system core dump was produced by a panic or a trap, copies of all the registers at the time of the trap or panic were saved in memory and are available in the core dump. For a trap, the registers are saved on the stack, in the order specified in the struct *save_state*, which is defined in */usr/include/machine/save_state.h*. For a panic, the registers are saved in a statically allocated memory location called *panic_save_state*, in the order specified in the struct *rpb*, which is defined in */usr/include/machine/rpb.h*. See the examples at the end of this chapter for details of how to access registers in the trap *save_state* area. The mechanics of accessing *panic_save_state* fields are similar, though the offsets into the save area are different. For example, if you want to get r3 out of the *panic_save_state* area, look at */usr/include/machine/rpb.h* and note that the field *rp_gr3* is the sixth word in struct *rpb*. Therefore, it can be found at *panic_save_state* + 5 words == *panic_save_state* + 0x14.

Not all registers in these save areas are guaranteed to be the same as at the time of the panic or trap, because some registers must be used by the system to execute the panic or trap path and save away the other registers. Registers which may not be preserved are r1, r19 - r22, r31, arg0, arg1, arg2, and arg3. Use your judgment with the contents of these registers in the save areas. If they look odd, they may have been overwritten.

If your stack trace includes a call to *trap()*, it will also have a call to *panic()* higher up (later in time) than the trap. In this case, it is safer to look in the trap *save_state* structure on the stack than the *panic_save_state* area for registers you are curious about, because the trap saved the registers closer in time to when the problem which caused the system crash occurred.

Obtaining Important Kernel Global Variables

To print out the value of a kernel global variable, simply use the symbol name with the appropriate formatting option (see adb(1) and the *ADB Tutorial* for more information). The following table lists some of the more interesting kernel globals, with the appropriate *adb* format for printing them, and brief descriptions of what they mean.

adb Command	Description
msgbuf+0xc/sD	Kernel's circular *printf* buffer.
freemem/D	Amount of free memory, in pages. If zero or a small number, system is out of memory.
physmem/D	Size of physical memory, in pages.
maxfree/D	Number of free pages soon after system boot.
desfree/D	Number of free pages the system tries to keep available.
minfree/D	Minimum free pages before system starts swapping processes out.
avefree/D	Average number of free pages over past 5 seconds.
avefree30/D	Average number of free pages over past 30 seconds.
freemem_cnt/D	Number of processes currently waiting for memory. If large number, many processes are stopped waiting for memory.
avenrun/3F	System load average, for the last one minute, five minutes, and 10 minutes, in floating point notation. If large numbers, system may be too heavily loaded.
lbolt/X	Seconds since boot.
time/Y	Current time, printed out in *ctime(3C)* format.
_release_version/s	HP-UX version string.
utsname+0x9/s	System hostname
utsname+0x12/s	HP-UX release number.
utsname+0x24/s	System hardware model number.

Obtaining Values from the Process Table Entry and User Area

It is possible to use *adb* to print out fields of interest from the process table entry and user area of the process that was running when the system crashed. The following subsection describes how to print certain important fields and gives a very brief description of each field. For more information on the meaning of these fields, see *The Design of the UNIX Operating System* by Maurice Bach, pub. Prentice-Hall, or *The Design and Implementation of the 4.3 BSD UNIX Operating System* by Leffler, McKusick, Karels and Quarterman, pub. Addison-Wesley.

adb, when called with the -k option, should print out the address of the user area and process table entry of the process that was running when the system crashed. *adb* will print this out when it is first entered, so the first output you should see from *adb* is:

```
u 7FFE6000 u.u_procp 4D2F20
```

u is the location of the user area, and should always be at virtual address 7FFE6000. When the kernel switches to a new process, it always maps the physical address of the process' user area to virtual address 7FFE6000. *u.u_procp* is the location of this process' process table entry. This address will vary from process to process. If *adb* does not print the *u* and *u.u_procp* values on entry, it was unable to determine the currently running process at crash time. *adb* was unable to print these values probably because your core dump was the result of a Transfer of Control (TOC).

If the process that caused the panic was running on the Interrupt Control Stack (ICS), the *u* and *u.u_procp* pointers will not contain valid information for the process. When an interrupt occurs the kernel executes the appropriate kernel code to process the interrupt without switching to a new user context. The *u* and *u_procp* address which *adb* will print will be the process that was running when the interrupt occurred. The interrupt interrupted the running of that process in order to process the interrupt. Look at the panic message in *msgbuf* to tell if the panic occurred while on the ICS. If you see a message like the following after the hex stack trace, the process was on the ICS.

```
NOT sync'ing disks (on the ICS) (0 buffers to flush):
```

Important User Area Fields

The table below describes the *adb* command to use to print important user area fields. *u* means the value marked *u* printed on *adb* entry (see example above). When executing the *adb* commands in the table below, substitute the *u* value printed on *adb* entry for the letter *u*.

Field Name	Address	Description
u_procp	u+0x258/X	Pointer to process table entry.
u_comm	u+0x260/s [Series 700] u+0x264/s [Series 800]	Name of command used to start this process. For STREAMS/UX, this is usually *strsched*.
u_arg	u+0x270/10X [Series 700] u+0x274/10X [Series 800]	Arguments to current system call. For STREAMS/UX service routines being run by *strsched*, these should all be zero.

For example, to print *u_comm*, given the *adb* entry printout u 7FFE6000 u.u_procp 4D2F20, type:

```
0x7FFE6000+0x260/s
```

See */usr/include/sys/user.h* for more information on fields in the user area. These offset values are for HP-UX release 10.0, and may change from release to release.

Important Process Table Fields

The table below describes the *adb* command to use to print important process table fields. *p* means the value marked *u.u_procp* printed on *adb* entry (see example above). When executing the *adb* commands in the table below, substitute the *u.u_procp* value printed out on *adb* entry for the letter *p*. For example, to print out *p_flag*, given the *adb* entry printout at the beginning of this section, type:

```
0x4D2F20+0x20/X
```

See */usr/include/sys/proc.h* for more information on fields in the proc structure. These offset values are for HP-UX release 10.0, and may change from release to release.

Field Name	Address	Description
p_flag	p+0x20/X [Series 700] p+0xc/X [Series 800]	per-process flags, see *proc.h*
p_flag2	p+0x24/X [Series 700] p+0x48/X [Series 800]	per-process flags, see *proc.h*
p_mpflag	p+0x10/X [Series 800 only]	per-process flags, see *proc.h*
p_stat	p+0xc/b [Series 700] p+0x32/b [Series 800]	current process state, see *proc.h*
p_uid	p+0x2c/D [Series 700] p+0x0x50/D [Series 800]	real user id, used to direct tty signals
p_suid	p+0x30/D [Series 700] p+0x54/D [Series 800]	set effective uid
p_pid	p+0x38/D [Series 700] p+0x5c/D [Series 800]	process id
p_ppid	p+0x3c/D [Series 700] p+0x60/D [Series 800]	process id of parent
p_pgrp	p+0x34/D [Series 700] p+0x58/D [Series 800]	process id of process group leader
p_wchan	p+0x40/X [Series 700] p+0x1c/X [Series 800]	event process is sleeping on should be zero if currently running
p_sleeptime	p+0x24/X [Series 800 only]	time of last sleep or wakeup (in seconds)
p_cptickstotal	p+0x4c/X [Series 700] p+0x14/X [Series 800]	cpu ticks (total for life of process)
p_cursig	p+0xe/b [Series 700] p+0x34/b [Series 800]	number of current pending signal, if any
p_sig	p+0x10/X [Series 700] p+0x38/X [Series 800]	signals pending to this process
p_sigmask	p+0x14/X [Series 700] p+0x3c/X [Series 800]	current signal mask

Field Name	Address	Description
p_sigignore	p+0x18/X [Series 700] p+0x40/X [Series 800]	signals being ignored
p_sigcatch	p+0x1c/X [Series 700] p+0x44/X [Series 800]	signals being caught by user

Debugging Examples

Example 1

The following core dump was obtained while using a modified version of the *sp* driver, which is described in example #2 in the *strdb* section of this chapter.

On entry to *adb*, we first look at the *msgbuf* to look for the panic message and hex stack trace. The interesting portion of *msgbuf* for this dump is:

```
msgbuf+10/s
        .
        .
        .
interrupt type 15, pcsq.pcoq = 0.3b2cc, isr.ior = 0.0
Data page fault on interrupt stack
   B2352A HP-UX () #1: Fri Aug 14 00:49:59 PDT 1992
panic: (display==0xbf00, flags==0x0) Interrupt
PC-Offset Stack Trace (read across, most recent is 1st):
  0x0013e81c   0x000cddb8   0x000bc93c   0x0003b2cc   0x0012e2bc
0x0016b350
End Of Stack
```

First we translate the hex stack trace in the panic message into procedure names and addresses. Using the *adb i* command for each of the hex addresses in the panic message stack trace, we get the following symbolic stack trace:

```
panic+40:       addil   800,dp
interrupt+7E8:  rsm     1,r0
$ihndlr_rtn:    rsm     1,r0
sp_timeout+2C:  ldws    0(arg3),arg2
softclock+94:   b,n     softclock+30
external_interrupt+350:         ldil    261000,r22
```

The address where the illegal data access occurred is *sp_timeout*+2C. The *isr.ior* in the panic message indicates that the data address that caused the panic is 0.0, and the instruction at *sp_timeout*+2C is *ldws* 0(*arg3*),*arg2*, so *arg3* must have been 0 at the time of the panic. So we are probably dereferencing a null pointer. Our first task is to find out which pointer this is. To do this we need to know which source code line *sp_timeout*+2C corresponds to. Here is the source code for *sp_timeout()*:

```
struct sp {
  unsigned sp_state;
  queue_t  *sp_rdq;
  mblk_t   *mp;
  mblk_t   *last_mp;
};

static sp_timeout(lp)
struct sp *lp;
{
  mblk_t *temp;
  unsigned int s;

  if (lp->sp_state & SPOPEN) {
    /* Put message on driver's read queue */
      s = splstr();
      temp = lp->mp;
      lp->mp = lp->mp->b_next;
      if (lp->mp == NULL) lp->last_mp = NULL;
      temp->b_next = NULL;
      putq(lp->sp_rdq,temp);
      splx(s);
  }
}
```

Here is the relevant portion of the assembly code. The instruction which caused the panic is marked with an "*."

```
 sp_timeout,20?ia                          # adb command
 sp_timeout:                               # adb's response
 sp_timeout:     stw     rp,-14(sp)
 sp_timeout+4:   stwm    r3,40(sp)
 sp_timeout+8:   stw     r4,-3C(sp)
 sp_timeout+0xC: or      arg0,r0,r3
 sp_timeout+10:  ldws    0(r3),arg1
 sp_timeout+14:  bb,>=,n arg1,31,sp_timeout+58
 sp_timeout+18:  bl      tmxlwsrv+6C,rp  (splstr)
 sp_timeout+1C:  or      r0,r0,r0
 sp_timeout+20:  or      ret0,r0,r4
 sp_timeout+24:  ldws    8(r3),arg1
 sp_timeout+28:  ldws    8(r3),arg3
*sp_timeout+2C:  ldws    0(arg3),arg2
 sp_timeout+30:  stws    arg2,8(r3)
```

At *sp_timeout*+0xC, arg0, which corresponds to the source code variable *lp* is moved to r3. We know arg0 is *lp*, because *lp* is the first argument to *sp_timeout()*. *sp_timeout*+0x14 looks like the if statement in the source code, because *bb* is a branch instruction. *sp_timeout*+0x18 is the call to *splstr()*. *sp_timeout*+0x28 loads arg3 with the memory contents at location r3 + 0x8. arg3 is the source code variable *lp->mp*. We can guess this because mp is 8 bytes from the start of *lp*, according to the declaration for the struct *sp*. So our problem is that *lp->mp* is NULL. We want to confirm

this, and want to look at the rest of **lp*. To do so, we need to find the value of r3 at the time of the panic. We may be able to extract this information from the stack if we know the value of *sp* at the time of the panic. To get this information, we do a manual stack back-trace. See "Manual Stack Back-Tracing" for details on how this is done. The resulting table is shown below:

sp	pcoqh	Procedure Address	Frame Size
0x1fdb80	0x24b34	doadump+0xec	0x30
0x1fdb50	0xc8f48	panic_boot+0x354	0x80
0x1fdad0	0xc8938	boot+0x24	0x100
0x16860	0x13e8cc	panic+0xf0	0x80
0x167e0	0xcddb8	interrupt+0x7e8	0x280
0x16560	0xbc93c	$ihndlr_rtn	0x230
0x16330	0x3b2cc	sp_timeout+0x2c	0x40
0x162f0	0x12e2bc	softclock+0x94	0x80

Now that we have the values of *sp*, we want to look into the stack of the procedure above *sp_timeout()* in the stack trace to find what value that procedure saved in its stack for r3. In this case, the procedure above *sp_timeout()* is *$ihndlr_rtn*. *$ihndlr_rtn* is one of the low-level kernel utility procedures which is hand-coded in assembly and does not create a normal stack frame. Instead it creates a "save state" area, which contains the values of all the registers at the time the trap or interrupt took place. The structure *save_state* is defined in */usr/include/machine/save_state.h*. The general registers are stored first, and are located at "top of save state area" + "register number" * 4. For example, r3 will be 3*4 = 12 off of the beginning of the save state area. To find the top of the save state area, subtract the size of the *save_state* structure from the value of *sp* for *$ihndlr_rtn*:

```
0x16560-0x230>sp          # set sp to top of trap save state
<sp/X
16330:        0xF000009    # first word of save state area
<sp+0xC/X                  # find contents of r3 (lp) at sp + 3*4
icsBase+33C:  24C258
24C258+0x8/X               # find 8 off of r3 (lp->mp)
sp_sp+18:     0             # lp->mp is NULL
0x24c258/4X                # look at all of lp:
              # state *sp_rdq     *mp       *last_mp
sp_sp+10:     1         1040C00    0         10F7C00
```

We can also use *strdb* to look at *lp*. (See the *strdb* section of this chapter for details.) There may be several instances of the *sp* driver, each with a different minor number, so we must look at each one until we find the

instance whose *q_ptr* is the same as the address we have for *lp*. *lp* is a pointer to the *sp* driver's private data, which is also pointed to by *q_ptr*. The strdb STREAMS/UX subsystem *la* command will tell us what minor numbers are in use for the *sp* driver:

```
:la sp

sp MAJOR = 115
ACTIVE Minor  2  Stream head RQ = 0x0810eb000
ACTIVE Minor  1  Stream head RQ = 0x081107a00
ACTIVE Minor  0  Stream head RQ = 0x0810ebe00
```

The strdb STREAMS/UX subsystem command *lm* will show us what modules may have been pushed into the stream above the *sp* driver:

```
:lm sp 0

STREAM Head
lmodc
Driver sp
```

In this case, the panicking stream happens to correspond to the *sp* with minor number 1. From the *strdb* STREAMS/UX subsystem, we use ":qh sp 1" to get to the read queue of the stream head containing *sp* driver with minor number 1. Then the *o* command to get to the write queue of the stream head. Next the *n* command twice to get from the stream head through the module lmodc to the driver *sp*. Here is the display of the *q* information for driver *sp*, minor number 1. Note that *q_ptr* is 0x24c258, which is the address of *lp*.

```
:qh sp 1

struct queue 0x1040c74

q_qinfo   =  0x1e545c   q_pad1[2] = 00
q_first   =  0x0q_other   =  0x1040c00
q_last    =  0x0
q_next    =  0x0
q_link    =  0x0
q_ptr     =  0x24c258
q_count   =  0
q_flag    =  0x1128
   QWANTR
   QUSE
   QOLD
   QSYNCH
q_minpsz  =  0
q_maxpsz  =  256
q_hiwat   =  0x8000
q_lowat   =  0x4000
q_bandp   =  0x105fd40
```

```
q_nband   =  1
q_pad1[0] = 00
q_pad1[1] = 00
```

Now that we have reached the queue structure for the panicking *sp* driver instance, we can use *strdb* or *adb* to examine its contents. Using the *strdb* command *:b*, we can look at *q_ptr*, and see that its *mp* field (the third word) is NULL:

```
:b 0x24c258

0x0024c258 00 00 00 01 01 04 0c 00 00 00 00 01 0f 7c 00 | .............
0x0024c268 00 00 00 01 01 0f 8e 00 00 00 00 00 00 00 00 | .............
0x0024c278 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .............
0x0024c288 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .............
0x0024c298 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .............
0x0024c2a8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .............
0x0024c2b8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .............
0x0024c2c8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .............
0x0024c2d8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .............
0x0024c2e8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .............
0x0024c2f8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .............
0x0024c308 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .............
0x0024c318 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .............
0x0024c328 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .............
0x0024c338 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .............
```

At this point, we have probably learned all that we can from the dump and must turn to the source code to discover the cause of this problem. We next examine the code carefully everywhere that *lp->mp* is updated or should be updated. Because *sp* driver's put routine, *spput()*, should be updating *lp->mp*, we look at it first.

```
static spput(q, mp)
queue_t *q;
mblk_t *mp;
{
  struct sp *lp;
  unsigned int s;

  switch (mp->b_datap->db_type) {
  case M_DATA:
  case M_PROTO:
  case M_PCPROTO:
    s = splstr();
    lp = q->q_ptr;
    if (!lp->last_mp)
      lp->last_mp = mp;
    else
      lp->last_mp->b_next = mp;
    splx(s);
```

```
    timeout(sp_timeout,lp,1);
    break;
  default:
    printf("Routine spput: Should not be here\n");
    break;
  }
}
```

Note that *spput()* never updates *lp->mp*. It just adds the new message to the tail of the list using *lp->last_mp*. But once *sp_timeout()* has processed the last message on the list and set *lp->mp* to NULL, *spput()* will never update *lp->mp* to point at the next message it receives. This causes *sp_timeout()* to be called with *lp->mp* == NULL. If we change *spput()* if statement to properly update *lp->mp* as shown below, this panic will be fixed.

```
if (!lp->mp)
        /*
       * head of list is NULL so list is empty -- put new message
         * at head of list
         */
        lp->mp = mp;
else
        /*
         * list is not empty -- put new message at tail of list
         */
        lp->last_mp->b_next = mp;
/*
 * update list tail pointer to point to new message
 */
lp->last_mp = mp;
```

Example 2

The following core dump was obtained while using a modified version of the *sp* driver, which is described in example #2 in the *strdb* section of this chapter.

On entry to *adb*, we first look at the *msgbuf* to look for the panic message and hex stack trace. The interesting portion of *msgbuf* for this dump is:

```
msgbuf+0xc/s
        .
        .
        .
trap type 15, pcsq.pcoq = 0.3b584, isr.ior = 0.0
   B2352A HP-UX () #1: Fri Aug 14 00:49:59 PDT 1992
panic: (display==0xbf00, flags==0x0) Data segmentation fault
PC-Offset Stack Trace (read across, most recent is 1st):
  0x0013e81c  0x000cc108  0x000bd3f4  0x0003b584  0x00049a48  0x0004bd0c
  0x0002f7d4  0x00046178  0x00049a48  0x000460d0  0x00046594  0x0012cc10
   0x000bedd0  0x00024cf0
End Of Stack
```

First we translate the hex stack trace in the panic message into procedure names and addresses. Using the *adb i* command for each of the hex addresses in the panic message stack trace, we get the following symbolic stack trace:

```
panic+40:              addil   800,dp
trap+0xA28:            b       trap+0xF18
$call_trap+20:         rsm     1,r0
spput+4C:              stws    r31,0(r1)
csq_lateral+80:        b,n     csq_lateral+8C
puthere+4C:            ldw     -54(sp),rp
lmodcsrv+5C:           bl      getq,rp
sq_wrapper+50:         ldw     -54(sp),rp
csq_lateral+80:        b,n     csq_lateral+8C
runq_run+58:           b,n     runq_run+74
str_sched_daemon+264: b        str_sched_daemon+160
```

The address where the illegal data access occurred is *spput+4C*. The *isr.ior* in the panic message indicates that the data address that caused the panic is 0.0, and the instruction at *spput+4C* is stws r31,0(r1), so r1 must have been 0 at the time of the panic. We are probably dereferencing a null pointer. Our first task is to find out which pointer this is. To do this we need to know to which source code line *spput+4C* corresponds to. Here is the source code for *spput()*:

```
struct sp {
  unsigned sp_state;
  queue_t  *sp_rdq;
  mblk_t   *mp;
  mblk_t   *last_mp;
};

static spput(q, mp)
queue_t *q;
mblk_t *mp;
{
  struct sp *lp;
  unsigned int s;

  switch (mp->b_datap->db_type) {
  case M_DATA:
  case M_PROTO:
  case M_PCPROTO:
    lp = q->q_ptr;
    if (!lp->mp)
      lp->mp = mp;
    else
      lp->last_mp->b_next = mp;
    lp->last_mp = mp;
    timeout(sp_timeout,lp,1);
    break;
  default:
```

```
    printf("Routine spput: Should not be here\n");
    break;
  }
}
```

Here is the relevant portion of the assembly code. The instruction where the panic occurred is marked with an "*".

```
  spput,40?ia
  spput:
  spput:          stw     rp,-14(sp)
  spput+4:        ldo     40(sp),sp
  spput+8:        or      arg1,r0,r31
  spput+0xC:      ldw     14(r31),r22
  spput+10:       ldbs    0xD(r22),arg1
  spput+14:       ldo     -41(r0),arg2
  spput+18:       ldo     -41(arg1),arg3
  spput+1C:       combt,=,n       arg2,arg3,spput+30
  spput+20:       ldo     -40(r0),ret1
  spput+24:       combt,=,n       ret1,arg3,spput+30
  spput+28:       ldo     42(r0),r19
  spput+2C:       combf,=,n       r19,arg3,spput+78
  spput+30:       ldw     14(arg0),arg1
  spput+34:       ldws    8(arg1),ret0
  spput+38:       comibf,=,n      0,ret0,spput+48
  spput+3C:       stws    r31,8(arg1)
  spput+40:       b       spput+54
  spput+44:       stws    r31,0xC(arg1)
  spput+48:       ldws    0xC(arg1),r1
* spput+4C:       stws    r31,0(r1)
  spput+50:       stws    r31,0xC(arg1)
  spput+54:       ldil    3B000,rp
  spput+58:       ldo     298(rp),r20
  spput+5C:       extru,= r20,1F,1,r21
  spput+60:       ldw     -4(dp),r21
  spput+64:       ldo     1(r0),arg2
  spput+68:       bl      spclose+0xB4,rp  (timeout)
```

First, we try to get a general idea where *spput+0x4C* falls in the source code. It occurs before the call to *timeout()* at *spput*+0x68. The pattern of *combt* and *combf* instructions from *spput*+0x1C to *spput*+0x2C correspond to the switch statement in the source code. We guess this by noticing that we have loaded a value into arg3 which we compare against three different values, which resembles the first three case statements in the switch statement. It is unlikely that the default case of the switch statement, which just does a *printf()*, would cause the system to panic. *spput*+0x4C is probably in the source code in the case statement for M_DATA, M_PROTO, and M_PCPROTO. The *comibf* instruction at *spput*+0x38 must correspond to the if (!lp->mp) source statement, because it is a conditional branch statement, and it is comparing a register to 0 (zero).

We may be able to determine whether we executed the “if” clause or the “else” clause of the if statement, based on the fact that we know we executed spput+0x4C (because a trap occurred while executing it). The *comibf* instruction branches to its target address if the condition it is checking is false. This *comibf* instruction compares ret0 to zero. If ret0 equals zero, *comibf* will not branch, and execution will continue to *spput*+0x3C and *spput*+0x40. *spput*+0x40 is an unconditional branch to *spput*+0x54, which is past *spput*+0x4C. Therefore, if ret0 had been zero, we never would have executed *spput*+0x4C. So ret0 was not zero. Since we know that the *comibf* instruction corresponds to “if (!lp->mp),” we know that lp->mp was not NULL, and the *comibf* instruction branches to *spput*+0x48 if lp->mp is not NULL, we can be confident that *spput*+0x48 and *spput*+0x4C are part of the *else* clause of the *if* statement, which consists of one statement, “lp->last_mp->b_next = mp;.”

Now we know which source code line we panicked on. We need to determine which source code pointer the register r1 corresponds to, because dereferencing r1 is what caused the panic. To do this, we work backwards from *spput*+0x4C to see where r1's contents came from. On *spput*+0x48, r1 gets loaded from arg1 + 0xC. Now we look backward to see where arg1 came from. It is tempting to assume that arg1 is the second argument to *spput*, which is *mp*. But at *spput*+0x10, arg1 is the target of a load, so at *spput*+0x48 arg1 does not contain mp. It is also tempting to look at *spput*+0x44 for the origins of arg1's contents, because that instruction has arg1 as its target. But because we took the *comibf* at *spput*+0x38, we must have branched around *spput*+0x44, so we can ignore this instruction. Looking further backward to *spput*+0x30, arg1 gets loaded from arg0 + 0x14. arg0 has not been the target of a load instruction since the beginning of *spput*, so it must still contain the first argument to *spput*, *q*. Looking at the source code, the only time that *q* is referenced is to set *lp* in the statement before the *if*. So arg1 must correspond to *lp*. Looking at the source code line where the panic occurred, “lp->last_mp->b_next = mp,” and the assembly code lines *spput*+0x48 and *spput*+0x4C, it appears that *spput*+0x48 is setting r1 to lp->last_mp, and *spput*+0x4C is attempting to put the contents of r31 into memory location r1 + 0, which must be “lp->last_mp->b_next”.

So our problem is that lp->last_mp is NULL. It may help us to look at the rest of **lp*, and to do so we need to find the value of arg1 at the time of the panic. We may be able to extract this information from the stack if we know

the value of sp at the time of the panic. To get this information, we do a manual stack back-trace. See “Manual Stack Back-Tracing” for details on how this is done. The resulting table is shown below:

sp	pcoqh	Procedure Address	Frame Size
0x1fdb80	0x24b34	doadump+0xEC	0x30
0x1fdb50	0xc8f48	panic_boot+354	0x80
0x1fdad0	0xc8938	boot+0x24	0x80
0x7ffe6f88	0x13e8cc	panic+0xf0	0x80
0x7ffe6f08	0xcc108	trap+0xf18	0x100
0x7ffe6e08	0xbd3f4	$call_trap	0x230
0x7ffe6bd8	0x3b584	spput+0x4c	0x40
0x7ffe6b98	0x49a48	csq_lateral+0x80	0x80

Now that we have the values of *sp*, we want to look into the stack frame of the procedure above *spput()* in the stack trace, to find what value that procedure saved in its stack for arg1. In this case, the procedure above *spput()* is *$call_trap*. *$call_trap* is one of the low-level kernel utility procedures which is hand-coded in assembly and does not create a normal stack frame. Instead it creates a “save state” area, which contains the values of all the registers at the time the trap or interrupt took place. The structure *save_state* is defined in */usr/include/machine/save_state.h*. The general registers are stored first, and are located at “top of save state area” + “register number” * 4. So, for example, arg1, which is also known as r25, will be 25*4 = 100 off of the beginning of the save state area. To find the top of the save state area, subtract the size of the *save_state* structure (0x230 in release 9.0) from the value of *sp* for *$call_trap*:

```
0x7ffe6e08-0x230>sp     # set sp to top of trap save state
<sp/X
7FFE6BD8:     0xF000009  # first word of save state area
0x7ffe6bd8+0x4/X
7FFE6BDC:     0          # find contents of r1 (lp->last_mp)
                         # at sp + 1*4.  NULL, as we thought
0x7ffe6bd8+0x64/X        # find contents of arg1 (lp) at
                         # sp + 25*4.
7FFE6C38:     0xFFFFFFBF
```

0xFFFFFFBF is a very unlikely value for *lp*. It is more likely that the contents of arg1 were changed in the process of taking a trap. The four arg registers are considered scratch registers, and the trap path is very likely to have overwritten these registers before it created the save state area.

However, there is an alternative way to find out the value of *lp*. If we can determine what the procedure that called *spput()* set arg0 to before the call, we will know the value of *q*, and *lp* is q->q_ptr.

The procedure which called *spput()* is *csq_lateral()*. The point where the call was made is marked with an asterisk. Note that the procedure call here is made using the instruction *ble* instead of the usual instruction *bl*. This is because *csq_lateral* does not know the name of the procedure it is going to call. *csq_lateral()* is passed a structure which contains the address of a procedure to call and the arguments with which to call it. Because the compiler cannot tell at compile time how far away in the executable image the procedure address is, it must use a branch and link external, *ble*, instruction in order to be sure it will be able to reach the procedure address being branched to.

```
  csq_lateral+40,15?ia
  csq_lateral+40: ldws    8(r3),arg2
  csq_lateral+44: depi    -1,1E,1,arg2
  csq_lateral+48: stws    arg2,8(r3)
  csq_lateral+4C: bl      csq_turnover+108,rp  (UNCRIT)
  csq_lateral+50: or      r6,r0,arg0
  csq_lateral+54: ldw     10(r5),ret1
  csq_lateral+58: comibt,=,n      0,ret1,csq_lateral+68
  csq_lateral+5C: ldw     10(r5),arg0
  csq_lateral+60: ldw     1C(arg0),r19
  csq_lateral+64: bb,<,n  r19,18,csq_lateral+84
  csq_lateral+68: ldw     1C(r5),arg1
  csq_lateral+6C: stw     r0,1C(r5)
  csq_lateral+70: ldw     14(r5),r6
  csq_lateral+74: ldw     18(r5),arg0
* csq_lateral+78: ble     0(sr4,r6)
  csq_lateral+7C: or      r31,r0,rp
  csq_lateral+80: b,n     csq_lateral+8C
```

At *csq_lateral*+0x74, arg0 is loaded from r5 + 0x18. So if we can find out what value r5 had at that point, we can determine the value of *q*. r5 is a callee save register, so there is a chance that *spput* saved r5 in its stack frame. We look at the first few instructions of *spput*:

```
spput/6i
spput:
spput:          stw     rp,-14(sp)
                ldo     40(sp),sp
                or      arg1,r0,r31
                ldw     14(r31),r22
                ldbs    0xD(r22),arg1
                ldo     -41(r0),arg2
```

We see that *spput* did not save r5. Callee registers are only saved if the callee plans to overwrite the register. So we cannot get r5 from *spput*'s stack frame, but if *spput* did not save r5 that means it did not overwrite it; therefore, the value for r5 in the save state area will be the same value that r5 had at *csq_lateral*+0x74. Look at 4*5 into the save state area:

```
<sp+0x14/X              # sp + 4*5 == r5
7FFE6BEC:       11002A0
11002A0+0x18/X          # q is r5 + 0x18
11002B8:        10EE674
10EE674+0x14/X          # lp is q + 0x14
10EE688:        24C278
24C278+0xC/X            # lp->last_mp = lp + 0xC
sp_sp+3C:       0       # lp->last_mp is NULL
0x24c278/4X             # look at all of lp:
                # state sp_rdq        mp        last_mp
sp_sp+30:       1       10EE600       0          0
```

Note that at the point the panic occurred, *lp->mp* was NULL, even though we can be sure that at the time we checked *lp->mp* at instruction *spput*+0x38, lp->mp was not NULL. How can this be true? As we saw in the previous example, *sp_timeout()* modifies the *lp* structure, and it runs out of timeout. In other words, *spput()* calls *timeout()* to schedule *sp_timeout()* to run after a specified amount of time. At each system clock tick, the kernel examines the list of procedures created by *timeout()* and schedules those procedures whose time has expired to run. Because a clock tick is a high level interrupt, it can occur at any time, and may suspend *spput()* if it is running. A clock tick may have occurred between *spput*+0x38 and *spput*+0x4C, allowing *sp_timeout()* to run and set *lp->mp* to NULL. In order to prevent this, we need to protect access to the *lp* structure by using *splstr()* around all critical sections of code in the *sp* driver which manipulate *lp*. So *spput()* source code should be changed as shown below:

```
case M_DATA:
case M_PROTO:
case M_PCPROTO:
  /*
   * Use splstr() to protect access to q->q_ptr area from
   * interrupts which may schedule sp_timeout().
   */
  s = splstr();
  lp = q->q_ptr;
  if (!lp->mp)
    lp->mp = mp;
  else
   lp->last_mp->b_next = mp;
  /*
   * Return to previous interrupt level
```

```
   */
  splx(s);
```

In order to protect access to q->q_ptr, *sp_timeout()* must also call *splstr()* before it accesses q->q_ptr. The source code for *sp_timeout()* in the first example in this section shows the correct use of *splstr()*.

See the STREAMS/UX synchronization section of Chapter 3 for guidelines on protecting module and driver critical sections.

Example 3

The following core dump was obtained while using a modified version of the *sp* driver, which is described in example #2 in the *strdb* section of this chapter.

On entry to *adb*, we first look at the *msgbuf* to look for the panic message and hex stack trace. The interesting portion of *msgbuf* for this dump is:

```
msgbuf+0xc/s
        .
        .
        .
trap type 15, pcsq.pcoq = 0.9ea14, isr.ior = 0.d
@(#)9245XA HP-UX (A.09.00) #0: Thu Aug 13 23:17:54 PDT 1992
panic: (display==0xbf00, flags==0x0) Data segmentation fault

PC-Offset Stack Trace (read across, most recent is 1st):
 0x0016da70 0x000e5a68 0x000d34cc 0x0009ea14 0x00099714 0x0009
2fdc
 0x0006e0c8 0x0006dbb8 0x0006d2a8 0x001954e8 0x00194fa4 0x000b
7e24
 0x001846d4 0x00181730 0x00156538 0x00156af8 0x001567b8 0x000e
6d80
 0x000d3aac
End Of Stack
```

First we translate the hex stack trace in the panic message into procedure names and addresses. Using the *adb i* command for each of the hex addresses in the panic message stack trace, we get the following symbolic stack trace:

```
panic+30:             addil   -1000,dp
trap+0xADC:           b       trap+1004
$call_trap+20:        rsm     1,r0
flushq+60:            ldbs    0xD(r21),r22
q_free+1C:            ldw     -0xA4(sp),r31
osr_pop_subr+0xB44:   b       osr_pop_subr+0xB4C
osr_close_subr+4D8:   stw     ret0,-40(sp)
pse_close+8A0:        stw     ret0,-3C(sp)
```

```
hpstreams_close+58:  stw     ret0,-40(sp)
call_open_close+448: or      ret0,r0,r3
closed+138:          or      ret0,r0,r5
ufs_close+11C:       movb,tr r0,ret0,ufs_close+15C
vn_close+24:         ldw     -54(sp),rp
vno_close+50:        addil   -59800,dp
closef+0xE8:         ldw     18(r3),arg0
exit+2B4:            bl      uffree,rp
rexit+20:            ldw     -54(sp),rp
syscall+2A4:         ldhs    0(r9),r19
```

The address where the illegal data access occurred is *flushq*+0x60. The *isr.ior* in the panic message indicates that the data address that caused the panic is 0.d, and the instruction at *flushq*+0x60 is ldbs 0xD(r21),r22, so r21 must have been 0 at the time of the panic. So we are probably dereferencing a null pointer. Our first task is to find out which pointer this is. To do this we need to know which variable r21 was supposed to contain. We do not have source code for *flushq()*, because it is a STREAMS/UX internal procedure, but we do know from its man page what arguments it takes, and we do have the assembly version of the code. Here is the relevant portion of the assembly. The instruction where the panic occurred is marked with an "*".

```
  flushq,20?ia
  flushq:
  flushq:          stw     rp,-14(sp)
  flushq+4:        ldo     40(sp),sp
  flushq+8:        stw     arg0,-64(sp)
  flushq+0xC:      stw     arg1,-68(sp)
  flushq+10:       ldw     -68(sp),r20
  flushq+14:       zdepi   1,10,1,r21
  flushq+18:       and     r20,r21,r22
  flushq+1C:       stw     r22,-3C(sp)
  flushq+20:       ldw     -68(sp),r31
  flushq+24:       addil   -8000,r0
  flushq+28:       ldo     -1(r1),r19
  flushq+2C:       and     r31,r19,r20
  flushq+30:       stw     r20,-68(sp)
  flushq+34:       ldw     -64(sp),r21
  flushq+38:       ldws    4(r21),r22
  flushq+3C:       stw     r22,-34(sp)
  flushq+40:       ldw     -34(sp),r1
  flushq+44:       comibt,=,n      0,r1,flushq+120
  flushq+48:       or      r0,r0,r0
  flushq+4C:       ldw     -34(sp),r31
  flushq+50:       ldws    0(r31),r19
  flushq+54:       stw     r19,-38(sp)
  flushq+58:       ldw     -34(sp),r20
  flushq+5C:       ldw     14(r20),r21
* flushq+60:       ldbs    0xD(r21),r22
  flushq+64:       stw     r22,-40(sp)
  flushq+68:       ldw     -68(sp),r1
```

We can find *flushq()*'s calling sequence in its man page in SVR4PG:

void flushq(queue_t *q, int flag)

It is more likely that **q* or one of its members is NULL than the parameter flag being the cause of our problem. We will trace the use of the first argument, originally in arg0, through *flushq*, to see how it might be related to the contents of r21.

At *flushq*+0x8, arg0 is pushed onto the stack at offset sp - 0x64. Neither arg0 nor -64(*sp*) is referenced again until *flushq*+0x34. At *flushq*+0x34, r21 is loaded with -64(*sp*), so at this point r21 contains **q*. At *flushq*+0x38, r22 is loaded from memory location 4 + r21. Looking at the structure definition for queue_t, found in */usr/include/sys/stream.h*, we see that the second word in a queue_t structure, which would be found at memory location r21 + 4, is the *q_first* pointer.

```
struct queue {
    struct qinit * q_qinfo;  /* procedures and limits for queue */
    struct msgb *  q_first;  /* head of message queue */
    struct msgb *  q_last;   /* tail of message queue */
    struct queue * q_next;   /* next QUEUE in Stream */
    struct queue * q_link;   /* link to scheduling queue */
    caddr_t        q_ptr;    /* to private data structure */
    ulong          q_count;  /* weighted count of characters on q
*/
    ulong          q_flag;   /* QUEUE state */
    long           q_minpsz; /* min packet size accepted */
    long           q_maxpsz; /* max packet size accepted */
    ulong          q_hiwat;  /* high water mark, for flow control
*/
    ulong          q_lowat;  /* low water mark */
    struct qband * q_bandp;   /* band information */
    unsigned char  q_nband;   /* number of bands */
    unsigned char  q_pad1[3]; /* reserved */
    struct queue * q_other;   /* pointer to other Q in queue pair
*/
    QUEUE_KERNEL_FIELDS
};
```

So r22 now contains *q->q_first*. At *flushq*+0x3C, r22 is stored back in the stack, at *sp* - 0x34.

At this point, it may be useful to try and work backwards from *flushq*+0x5C, where r21 gets loaded from 0x14 + r20, because at the next instruction, *flushq*+0x60, we know that r21 is NULL. We notice that at *flushq*+0x58, r20 is loaded from *sp* - 0x34. At *flushq*+0x3C, we know that *sp* - 0x34 was

q->q_first. Checking the instructions between *flushq*+0x3C and *flushq*+0x58 shows that *sp* - 0x34 has not been stored to by any of these instructions, only loaded from. So at *flushq*+0x58, r20 is loaded with *q->q_first*. At *flushq*+0x5C, r21 is loaded with some field of *q->q_first*. Looking at the structure definition for *struct msgb*, also found in */usr/include/sys/stream.h*, we find that the sixth word of the *msgb* structure, which would be found at memory location r20 + 5 words == r20 + 0x14, is *b_datap*.

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

So our problem is that *q->q_first->b_datap* is NULL. We want to confirm this, and to look at the rest of the *q* structure. To do that we need to find the value of *sp* - 0x64 at the time of the panic. We may be able to extract this information from the stack if we know the value of *sp* at time of the panic. To get this information, we do a manual stack back-trace. See “Manual Stack Back-Tracing” for details on how this is done. The resulting table is shown below:

sp	pcoqh	Procedure Address	Frame Size
0x2418c0	0x1c374	doadump+0xec	0x30
0x241890	0xdfcd0	panic_boot+0x354	0xc0
0x2417d0	0xdf3a8	boot+0x34	0x80
0x7ffe7750	0x16db14	panic+0xd4	0x40
0x7ffe7710	0xe5a68	trap+0xadc	0xc0
0x7ffe7650	0xd34cc	$call_trap	0x230
0x7ffe7420	0x9ea14	flushq+0x60	0x40

Now that we have the values of *sp* for *flushq*, we know the *q* address we are interested in is at 0x7ffe7420 - 0x64:

```
0x7ffe7420-0x64/X
7FFE73BC:       5E9C00
```

Looking at the first few words of the *q* structure, we can determine the value of *q_first*, which is the second word:

```
5E9C00/4X
5E9C00:         294160          5D8C00          6C1880          0
```

Looking at *q_first*, we can see that the sixth word, *b_datap*, is NULL:

```
5D8C00/8X
5D8C00:         646480      0             646400         644000
                6440D1      0               0              0
```

We can also use *strdb* to look at *q* and *q_first*. See the *strdb* section of this chapter for more information. Because there may be several instances of the *sp* driver, each with a different minor number, we must look at each one until we find the stream which contains a queue whose address is the same as the address we have for *q*. The strdb STREAMS/UX subsystem *la* command will tell us what minor numbers are in use for the *sp* driver:

```
:la sp

sp MAJOR = 115
ACTIVE Minor 0x000013  Stream head RQ = 0x00607b00
ACTIVE Minor 0x000012  Stream head RQ = 0x00605c00
```

These instances of *sp* are far fewer than we had expected. *lm* on minor number 0x12 shows that *lmodc* has already been popped off the stream:

```
:lm sp 0x12

STREAM Head
Driver sp
```

and using :qh sp 0x12, and o and n as needed to traverse all the queues in this stream shows that none of these queues have address 0x5e9c00. *lm* on sp 0x13 shows that lmodc is still pushed above *sp* on this stream, but traversing all the queues in this stream shows that none of them are the queue we are looking for. We can use the *strdb* primary mode *:x* command to format *q_first* as a struct msgb to confirm our finding from *adb* that *q->q_first->b_datap* is NULL. (We find the structure type for *q_first* from */usr/include/sys/stream.h*).

```
:x msgb 0x5d8c00

struct msgb 0x5d8c00                    S:1

b_next  =  0x646480
b_prev  =  0x0
b_cont  =  0x646400
b_rptr  =  0x644000
b_wptr  =  0x6440d1
b_datap =  0x0
b_band  =    0
b_pad1  = 00
b_flag  =  0x0
b_pad2  =  0
```

b_datap could be NULL because its resources have been freed, or it could be NULL because the data structure was corrupted in some way. To try to narrow this down, we want to look at the message buffer b_cont. If its *b_datap* is also NULL, the possibility of corruption becomes less likely. We can use :x msgb 0x646400 to format the *b_cont* field of *q->q_first*. It is easier, however, to see if there is a navigation key available for the *b_cont* field. "?" lists the available navigation keys:

```
navigation for structure msgb
'n'      = b_next (msgb)
'p'      = b_prev (msgb)
'm'      = b_rptr (b_rptr)
'c'      = b_cont (msgb)
'd'      = b_datap (datab)
```

Using the *c* navigation key, we see that *b_datap* for *b_cont* is also NULL. This makes it very likely that this message has already been freed.

```
struct msgb 0x646400            S:2

b_next  =  0x5d8c00
b_prev  =  0x0
b_cont  =  0x0
b_rptr  =  0x651400
b_wptr  =  0x6517e1
b_datap =  0x0
b_band  =  0
b_pad1  =  00
b_flag  =  0x0
b_pad2  =  0
```

Now we try to get information about the queue which was pointing to this message at the time of the panic. We use *:x* to format 0x5e9c00 as a queue structure to see what information it may still contain.

```
:x queue 0x5e9c00
```

```
struct queue 0x5e9c00                S:3

q_qinfo   =  0x294160   q_pad1[0] = 00
q_first   =  0x5d8c00   q_pad1[1] = 00
q_last    =  0x6c1880   q_pad1[2] = 00
q_next    =  0x0        q_other   =  0x5e9c74
q_link    =  0x0
q_ptr     =  0x0
q_count   =  24896
q_flag    =  0x1135
   QREADR
   QFULL
   QWANTW
   QUSE
   QOLD
   QSYNCH
q_minpsz  =  0
q_maxpsz  =  256
q_hiwat   =  0x8000
q_lowat   =  0x4000
q_bandp   =  0x539d00
q_nband   =  1
```

Note that this is a read queue whose *q_next* pointer is NULL. This implies that this queue is not a connected part of a stream, and is in the process of being closed. To find out what driver or module this queue is being used by, we want to look at *q_qinfo*. We could use :x qinit 0x294160, or look for an appropriate navigation key:

```
?

navigation for structure queue
'i'      = q_qinfo (qinit)
'm'      = q_first (msgb)
'z'      = q_last (msgb)
'n'      = q_next (queue)
'l'      = q_link (queue)
'b'      = q_bandp (qband)
'o'      = q_other (queue)
```

We use the *i* navigation key to print the following:

```
struct qinit 0x294160                S:4

qi_putp   =  0x785ac
qi_srvp   =  0x78794
qi_qopen  =  0x7841c
qi_qclose =  0x78490
qi_qadmin =  0x0
qi_minfo  =  0x294148
qi_mstat  =  0x0
```

Using the *adb i* command, we can find out the name of the *qi_putp* routine:

```
0x785ac/i
lmodcput:
lmodcput:       stw     rp,-14(sp)
```

This means the module *lmodc* was using the queue on which the panic occurred. We can double check this by looking at the *qi_minfo* structure in *strdb*. Again, we can either use :x module_info 0x294148, or we could see if there is a navigation key available for *qi_minfo*:

```
?

navigation for structure qinit
'i'      = qi_minfo (module_info)
's'      = qi_mstat (module_stat)
```

Using the *qinit i* navigation key to print the *module_info* structure:

```
struct module_info 0x294148             S:5

mi_idnum  =  0x3ec
mi_idname =  0x23a0a8
mi_minpsz =  0
mi_maxpsz =  256
mi_hiwat  =  0x8000
mi_lowat  =  0x4000
```

and using the *adb s* command to print *mi_idname* as a string:

```
0x23a0a8/s
lmcinfo+10:     lmodc
```

So we had the panic occur on an *lmodc* read queue which was in the process of being closed. Our stack trace confirms this. We are making the exit system call, close all open file descriptors and as part of process clean-up. The last close of a stream causes each module and driver to be popped and its resources freed, including its message buffers. Whenever a panic occurs which involves *b_datap* being NULL, the cause is usually that the buffer has already been freed but a pointer to it was not zeroed out, and a module or driver continues to access the buffer through this non-zeroed pointer. The best way to find the cause of this problem is to look through the source code for all calls to *freemsg()* or *freeb()*, and check that all pointers to the buffer being freed are zeroed out.

For the *sp* driver, we found that *spclose()* calls *freemsg()*:

```
static spclose(q)
queue_t *q;
{
  struct sp *lp;
  unsigned int s;
  mblk_t *mp, *t_mp;

  lp = (struct sp *) (q->q_ptr);
  /* Free messages queued by spput() on interim mesg queue. */
  s = splstr();
  mp = lp->mp;
  while (mp != NULL) {
    t_mp = mp;
    mp = mp->b_next;
    freemsg(t_mp);
  }
  splx(s);
  flushq(WR(q), 1);
  q->q_ptr = NULL;
}
```

freemsg() is called to free all messages held in the interim message queue in our private data area, but we do not zero out the pointers *lp->mp* or *lp->last_mp*, which point to the head and tail of the private interim queue. A call to *sp_timeout()* may still be pending in the timeout queue. When *sp_timeout()* is executed, because *lp->mp* is non-NULL, it will call *putq()* to pass *lp->mp* up to *sp*'s read queue, where *sp*'s service routine will call *putnext()* to put it in *lmodc*'s read queue. When *flushq()* is called on *lmodc*'s read queue, it tries to free this already freed message, causing a trap type 15 panic on the NULL *b_datap*. Adding the following code to *spclose()* will fix this problem:

```
        .
        .
        .
    freemsg(t_mp);
  }
  splx(s);
  /*
   * NULL out list pointers to insure the messages they point to
   * will not be freed twice.
   */
  lp->mp = NULL;
  lp->last_mp = NULL;
  flushq(WR(q), 1);
  q->q_ptr = NULL;
}
```

7

STREAMS/UX-NetTL Link

NetTL (Network Tracing and Logging facility) is the facility used by network drivers and modules to capture network error events or trace data. In HP-UX 10.0, a mechanism will enable STREAMS/UX to deliver log/trace messages to NetTL. Previously, STREAMS/UX had its own error logging and tracing facility.

This chapter describes the STREAMS/UX-NetTL link, which integrates the STREAMS/UX logging and tracing facility with NetTL. STREAMS/UX error and trace messages generated by *strlog()* or by *putmsg()* to a STREAMS/UX log driver can also be delivered to NetTL with the STREAMS/UX-NetTL link.

With the STREAMS/UX-NetTL link, a single common interface for network tracing and logging will exist. Also, STREAMS/UX logging can benefit from NetTL's powerful features like message filtering.

Implementation of the STREAMS/UX-NetTL link is transparent to strerr and strace users. These commands work just as before even when NetTL is running.

Mapping from STREAMS/UX Messages to NetTL Messages

Both STREAMS/UX error logging and event tracing messages are mapped to NetTL logging messages.

NetTL log class is determined by STREAMS/UX log messages' flags according to the following rule:

```
If (flags & SL_ERROR)                    NetTL log class
        then                              ------------------
            if (flags & SL_FATAL)  --->  DISASTER
            if (flags & SL_WARN)   --->  WARNING
            if (flags & SL_NOTE)   --->  INFORMATIVE
            otherwise              --->  ERROR
        else all messages          --->  INFORMATIVE
```

As a default, only DISASTER and ERROR messages are logged. You can change this setting by using the *nettlconf* command (see nettlconf(1M)).

STREAMS/UX Subsystem ID and Subformatter

Subsystem ID

STREAMS/UX subsystem ID used by NetTL is:

ID Name: STREAMS
ID Number: 129

Subformatter

The messages logged by the NetTL facility can be formatted to a readable form by the *netfmt* command (see netfmt(1M)). The STREAMS/UX subformatter can be used to filter messages on STREAMS/UX module ID and sub-ID.

The filter configuration file syntax for STREAMS/UX is the following:

STREAMS module_id sub_id

module_id and sub_id can be a decimal number or * as a wild card.

For example:

```
STREAMS  1  100
STREAMS  2  *
STREAMS  *  101
```

Quick Guide On How to Use NetTL for STREAMS/UX

- Check if NetTL is running.

  nettl -status

  NetTL will start running by default after the system boot (see nettl(1M) for more detail).

  If NetTL is running, you can check the log file name, STREAMS/UX subsystem ID, STREAMS/UX log classes, etc.

- If it is not running, a superuser needs to start NetTL.

  nettl -start

- NetTL can be stopped by a superuser.

  nettl -stop

- You can change the set of NetTL log classes you are interested in.

  By default, only DISASTER and ERROR messages are logged. A superuser can modify this default by using the nettlconf command (see nettlconf(1M)). Bit masks for turning on log classes are the following:

INFORMATIVE	1
WARNING	2
ERROR	4
DISASTER	8

  For example:

  - To log only DISASTER messages,

    nettlconf -id 129 -class 8

  - To log DISASTER, ERROR, and WARNING messages,

    nettlconf -id 129 -class 14

  - To verify your changes,

    nettlconf -status

  - To activate your changes, you need to restart NetTL.

- You can format and read the logged messages.

  netfmt -f /var/adm/nettl.LOG00

  The default error log file is /var/adm/nettl.LOG00.

- You can format and filter the logged messages.

  netfmt -f /var/adm/nettl.LOG00 -c filter_file

  The filter_file would look like:

  Example 1: To format only STREAMS DISASTER messages:

```
formatter   filter  subsystem   STREAMS
formatter   filter  class        DISASTER
```

  Example 2: To filter on time:

```
formatter   filter  time_from      12:34:56   1/1/94
formatter   filter  time_through   21:43:56   1/2/94
```

  Example 3: To filter on STREAMS module ID and sub-ID:

```
STREAMS      1       100
STREAMS      2       *
STREAMS      *       101
```

  Example 4: More complex example:

```
formatter    filter     subsystem     STREAMS
formatter    filter     class         DISASTER
formatter    filter     class         ERROR
formatter    filter     class         WARNING
formatter    filter     time_from     12:34:56   1/1/94
formatter    filter     time_through  21:43:56   1/2/94
STREAMS      1          100
STREAMS      2          *
STREAMS      *          101
```

Index

Index