HP-UX Reference

Section 7: Device (Special) Files
Section 9: General Information
Index

HP-UX 11i Version 3

Volume 10 of 10

Manufacturing Part Number : B2355-91026
E0207

Printed in USA

© Copyright 1983-2007 Hewlett-Packard Development Company LP.

Legal Notices

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

Warranty

The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

U.S. Government License

Confidential computer software. Valid license from HP required for possession, use, or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data forCommercial Items are licensed to the U.S. Government under vendor's standard commercial license.

Additional Copyright Notices

This document and the software it describes may also be protected under one or more of the following copyrights. Additional copyrights are acknowledged in some individual manpages.

© Copyright 1979, 1980, 1983, 1985-1993 The Regents of the University of California.

© Copyright 1980, 1984, 1986 Novell, Inc.

© Copyright 1985, 1986, 1988 Massachusetts Institute of Technology

© Copyright 1986-2000 Sun Microsystems, Inc.

© Copyright 1988 Carnegie Mellon University

© Copyright 1989-1991 The University of Maryland

© Copyright 1989-1993 The Open Software Foundation, Inc.

© Copyright 1990 Motorola, Inc.

© Copyright 1990-1992 Cornell University

© Copyright 1991-2003 Mentat, Inc.

© Copyright 1996 Morning Star Technologies, Inc.

© Copyright 1996 Progressive Systems, Inc.

Trademark Notices

Intel and Itanium are registered trademarks of Intel Corporation in the US and other countries and are used under license.

Java is a US trademark of Sun Microsystems, Inc.

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

OSF/Motif is a trademark of The Open Group in the US and other countries.

UNIX is a registered trademark of The Open Group.

X Window System is a trademark of The Open Group.

Preface

HP-UX is the Hewlett-Packard Company's implementation of a UNIX® operating system that is compatible with various industry standards. It is based on the System V Release 4 operating system and includes important features from the Fourth Berkeley Software Distribution.

The ten volumes of this manual contain the system reference documentation, made up of individual entries called **manpages**, named for the `man` command (see *man* (1)) that displays them on the system. The entries are also known as manual pages or reference pages.

General Introduction

For a general introduction to HP-UX and the structure and format of the manpages, please see the *introduction* (9) manpage in volume 9.

Section Introductions

The manpages are divided into sections that also have introduction (intro) manpages that describe the contents. These are:

intro (1)	*Section 1: User Commands* (A-M in volume 1; N-Z in volume 2)
intro (1M)	*Section 1M: System Administration Commands* (A-M in volume 3; N-Z in volume 4)
intro (2)	*Section 2: System Calls* (in volume 5)
intro (3C)	*Section 3: Library Functions* (A-M in volume 6; N-Z in volume 7)
intro (4)	*Section 4: File Formats* (in volume 8)
intro (5)	*Section 5: Miscellaneous Topics* (in volume 9)
intro (7)	*Section 7: Device (Special) Files* (in volume 10)
intro (9)	*Section 9: General Information* (in volume 10)
Index	*Index, All Volumes* (in volume 10)

Typographical Conventions

audit (5)	An HP-UX manpage reference. For example, *audit* is the name and *5* is the section in the *HP-UX Reference*. On the web and on the Instant Information CD, it may be a hyperlink to the manpage itself. From the HP-UX command line, you can enter "`man audit`" or "`man 5 audit`" to view the manpage. See *man* (1).
Book Title	The title of a book. On the web and on the Instant Information CD, it may be a hyperlink to the book itself.
`Command`	A command name or qualified command phrase.
`ComputerOutput`	Text displayed by the computer.
Emphasis	Text that is emphasized.
Emphasis	Text that is strongly emphasized.
`ENVIRONVAR`	The name of an environment variable.
[`ERRORNAME`]	The name of an error number, usually returned in the `errno` variable.
KeyCap	The name of a (usually) nonprinting keyboard key, such as **Ctrl-X** or **Tab**. Note that **Return** and **Enter** both refer to the same key.
`Replaceable`	The name for a value that you replace in a command or function, or information in a display that represents several possible values.
Term	The defined use of an important word or phrase.
`UserInput`	Commands and other text that you type.
`$`	User command prompt.
`#`	Superuser (`root`) command prompt.

Command Syntax

`Literal`	A word or character that you enter literally.
`Replaceable`	A word or phrase that you replace with an appropriate value.
`-`*`chars`*	One or more grouped command options, such as `-ikx`. The *`chars`* are usually a string of literal characters that each represent a specific option. For example, the entry `-ikx` is equivalent to the individual options `-i`, `-k`, and `-x`. The plus character (+) is sometimes used as an option prefix.
`-`*`word`*	A single command option, such as `-help`. The *`word`* is a literal keyword. The difference from `-`*`chars`* is usually obvious and is clarified in an Options description. The plus character (+) and the double hyphen (`--`) are sometimes used as option prefixes.
[ ]	The bracket metacharacters enclose optional content in formats and command descriptions.
{ }	The brace metacharacters enclose required content in formats and command descriptions.
|	The bar metacharacter separates alternatives in a list of choices, usually in brackets or braces.
...	The ellipsis metacharacter after a token (*abc*...) or a right bracket ([ ]...) or a right brace ({ }...) metacharacter indicates that the preceding element and its preceding whitespace, if any, may be repeated an arbitrary number of times.
...	Ellipsis is sometimes used to indicate omitted items in a range.

Function Synopsis and Syntax

HP-UX functions are described in a definition format rather than a usage format. The definition format includes type information that is omitted when the function call is actually included in a program.

The function syntax elements are the same as for commands, except for the options; see “Command Syntax” on page 7.

Function General Definition

The general definition form is:

type func (*type param* [, *type param*]...);

For example:

`int setuname ( const char *`*name* `, size_t` *namelen* `);`

Function Usage

The usage form is:

func (*param* [, *param*]...);

For example:

`setuname (` *name* [, *namelen*]... `);`

Revision History

Part Number	Release; Date; Format; Distribution
B2355-60130	HP-UX 11i Version 3; February 2007; one volume HTML; http://docs.hp.com and Instant Information.
B2355-91017-26	HP-UX 11i Version 3; February 2007; ten volumes PDF; http://docs.hp.com, Instant Information and print.
B2355-60127	HP-UX 11i Version 1; September 2005 Update; one volume HTML; http://docs.hp.com and Instant Information.
B2355-90902-11	HP-UX 11i Version 1; September 2005 Update; ten volumes PDF; http://docs.hp.com and print.
B2355-60105	HP-UX 11i Version 2; September 2004 Update; one volume HTML; http://docs.hp.com and Instant Information.
B2355-90839-48	HP-UX 11i Version 2; September 2004 Update; ten volumes PDF; http://docs.hp.com and print.
B2355-60103	HP-UX 11i Version 2; August 2003; one volume HTML; http://docs.hp.com and Instant Information.
B2355-90779-87	HP-UX 11i Version 2; August 2003; nine volumes PDF; http://docs.hp.com and print.
B9106-90010	HP-UX 11i Version 1.6; June 2002; one volume HTML; http://docs.hp.com and Instant Information.
B9106-90007	HP-UX 11i Version 1.5; June 2001; seven volumes HTML; http://docs.hp.com and Instant Information.
B2355-90688	HP-UX 11i Version 1; December 2000; nine volumes.
B2355-90166	HP-UX 11.0; October 1997; five volumes.
B2355-90128	HP-UX 10.X; July 1996; five volumes; online only.
B2355-90052	HP-UX 10.0; July 1995; four volumes.

Volume Ten
Table of Contents

Section 7
Section 9

Index

Volume Ten
Table of Contents

Table of Contents
Volume Ten

Section 7: Device (Special) Files

Section 9: General Information

Index: All Volumes

Section 7

Device (Special) Files

Section 7

Device (Special) Files

NAME

intro - introduction to device special files

DESCRIPTION

This section describes the device special files (DSFs) and hardware paths used to access HP peripherals and device drivers. The names of the entries are generally derived from the type of device being described (disk, tape, terminal, and so on.), not the names of the device special files or device drivers themselves. Characteristics of both the hardware device and the corresponding HP-UX device driver are discussed where applicable.

Device Types

Devices can be classified in two device access modes, *raw* and *block*. A raw or character-mode device, such as a line printer, transfers data in an unbuffered stream and uses a character device special file.

A block-mode device, as the name implies, transfers data in blocks by means of the system's normal buffering mechanism. Block devices use block device special files and may have a character device interface too.

Device File Naming Convention

A device special file name becomes associated with a device when the file is created, either automatically by the special file daemon **sfd**, or explicitly with the **insf**, **mknod**, or **mksf** command. When creating device special files, it is recommended that the following standard naming convention be used:

/dev/*subdir***/***class*#[*options*]

subdir An optional subdirectory for the device class (for example, **rdisk** for raw device special files for disks, **disk** for block device special files for disks, **rtape** for raw tape devices).

class The class of device, such as **tape**, **disk**, or **lan**.

\# The instance number assigned by the operating system to the device. Each class of device has its own set of instance numbers, so each combination of class and instance number refers to exactly one device.

options Further qualifiers, such as disk partition (**p**#), tape density selection for a tape device, or surface specification for magneto-optical media.

Naming conventions for each type of device are described in their respective manpage entries.

Legacy mass storage device special files have a different naming convention that encodes the hardware path; this is described in the *Device File Types (Mass Storage Devices)* section.

Hardware Paths

Hardware path information, as well as class names and instance numbers, can be derived from **ioscan** output; see *ioscan*(1M). There are three different types of paths to a device: *legacy hardware path*, *lunpath hardware path*, and *LUN hardware path*. All three are numeric strings of hardware components, notated sequentially from the system bus address to the device address. Each number typically represents the location of a hardware component on the path to the device.

The *legacy hardware path* is composed of a series of bus-nexus addresses separated by slash (**/**) characters, leading to a host bus adapter (HBA). Beneath the HBA, additional address elements are separated by period (**.**) characters. All the elements are represented in decimal. This is the format printed by default by the **ioscan** command for most devices. An example of a legacy hardware path is **0/0/2/0.1.7.0**.

The *lunpath hardware path* is used for mass storage devices, also known as logical units (LUNs). It is identical in format to a legacy hardware path, up to the HBA. Beneath the HBA, additional elements are printed in hexadecimal. The leading elements representing a transport-dependent target address, and the final element is a LUN address, which is a 64-bit representation of the LUN identifier reported by the target. This format is printed by the **ioscan** command when the **-N** option is specified. The string **0/2/1/0.0x50001fe1500170ac.0x4017000000000000** is an example of a lunpath hardware path.

Note that the address elements beneath the HBA may not correspond to physical hardware addresses; instead, the lunpath hardware path should be considered a *handle*, not a physical path to the device.

The *LUN hardware path* is a virtualized path that can represent multiple hardware paths to a single mass storage device. Instead of a series of bus-nexus addresses leading to the HBA, there is a virtual bus-nexus (known as the *virtual root node*) with an address of 64000. Addressing beneath that virtual root node consists of a virtual bus address and a virtual LUN identifier, delimited by slash (**/**) characters. The string **64000/0xfa00/0x22** is an example of a LUN hardware path.

As a virtualized path, the LUN hardware path is only a handle to the LUN, and does not represent the LUN's physical location; rather, it is linked to the LUN's World Wide Identifier (WWID). Thus, it remains the same if new physical paths to the device are added, if existing physical paths are removed, or if any of the physical paths changes. This LUN binding persists across reboots, but it is not guaranteed to persist across installations — that is, reinstalling a system or installing an identically configured system may create a different set of LUN hardware paths.

Device File Types (Mass Storage Devices)

Mass storage devices, such as disk devices and tape devices, have two types of device files, *persistent* device special files and *legacy* device special files. Both can be used to access the mass storage device independently, and can coexist on the same system.

A *persistent* device special file is associated with a LUN hardware path, and thus transparently supports agile addressing and multipathing. In other words, a persistent device special file is unchanged if the LUN is moved from one HBA to another, moved from one switch/hub port to another, presented via a different target port to the host, or configured with multiple hardware paths. Like the LUN hardware path, the binding of device special file to device persists across reboots, but is not guaranteed to persist across installations. The device special file name follows the standard naming convention above, and the minor number contains no hardware path information.

A *legacy* device special file is locked to a particular physical hardware path, and does not support agile addressing. Such a device special file contains hardware path information such as SCSI bus, target, and LUN in the device file name and minor number. Specifically, the *class* and *instance* portions of the device special file name indicate hardware path information and are in the format **c**#**t**#**d**# as follows:

c# The instance number assigned by the operating system to the interface card, in decimal. It is a decimal number with a range of 0 to 255. There is no direct correlation between instance number and physical slot number.

t# The target address on a remote bus (for example, SCSI address). It is a decimal number with a typical range of 0 to 15.

d# The device unit number at the target address (for example, the LUN in a SCSI device). It is a decimal number with a typical range of 0 to 7.

Note that the legacy naming convention supports a maximum of 256 external buses and a maximum of 32768 LUNs. Systems with mass storage devices beyond those limits will be unable to address them using legacy naming conventions.

Legacy device special files are deprecated, and their support will be removed in a future release of HP-UX.

Viewing Mass Storage

With the advent of persistent and legacy device special files, commands dealing with mass storage can choose between two *views* of the I/O system. A command presenting the *legacy* view uses legacy device special files and legacy hardware paths. The *agile* view uses persistent device special files, lunpath hardware paths, and LUN hardware paths.

Depending on the command, both views may be presented, or the choice of view may be controlled by a command option or an environment variable. For example, the **ioscan** command shows the legacy view by default, and switches to the agile view if the **-N** option is specified.

EXAMPLES

Example 1

The following is an example of a persistent device special file name:

/dev/disk/disk3

where **disk** indicates block disk access and **disk3** indicates device class *disk* and instance number 3. The absence of **p**# indicates access to the entire disk; see *disk*(7) for details.

Example 2

The following is an example of a legacy disk device special file name:

/dev/dsk/c0t6d0s2

where **dsk** indicates block disk access and **c0t6d0** indicates logical disk access at interface card instance 0, target address 6, and unit 0. The **s2** indicates access to section 2 of the disk.

Example 3

The following is an example of a persistent tape device special file name:

/dev/rtape/tape4QIC150

where **rtape** indicates raw magnetic tape, **tape4** indicates tape device instance number 4, and **QIC150** identifies the tape format as QIC150; see *mt*(7) for details.

WARNINGS

The support of legacy device special files is deprecated and will be removed in a future release of HP-UX.

SEE ALSO

insf(1M), ioscan(1M), lssf(1M), mksf(1M), mknod(1M), hier(5), introduction(9).

System Administration's Guide at **http://docs.hp.com**.

The Next Generation Mass Storage Stack whitepaper at:
http://docs.hp.com/en/netsys.html#Storage%20Area%20Management.

a

NAME

arp - Address Resolution Protocol

DESCRIPTION

ARP is a protocol used to dynamically map between DARPA Internet and hardware station addresses. It is used by all LAN drivers.

ARP caches Internet-to-hardware station address mappings. When an interface requests a mapping for an address not in the cache, ARP queues the message that requires the mapping, and broadcasts a message on the associated network requesting the address mapping if the **ether** encapsulation method has been enabled for the interface. If a response is provided, the new mapping is cached and any pending message is transmitted. ARP queues at most one packet while waiting for a mapping request to be responded to; only the most recently "transmitted" packet is kept.

To facilitate communications with systems that do not use ARP, **ioctl** calls are provided to enter and delete entries in the Internet-to-hardware station address tables.

Application Usage:

```
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <net/if.h>
#include <netinet/if_ether.h>
struct arpreq arpreq;

ioctl(s, SIOCSARP, (caddr_t)&arpreq);
ioctl(s, SIOCGARP, (caddr_t)&arpreq);
ioctl(s, SIOCDARP, (caddr_t)&arpreq);
```

Each **ioctl** call takes the same structure as an argument. **SIOCSARP** sets an ARP entry, **SIOCGARP** gets an ARP entry, and **SIOCDARP** deletes an ARP entry. These **ioctl** calls can be applied to any socket descriptor *s*, but only by the super-user. The **arpreq** structure contains:

```
/*
 * ARP ioctl request
 */
struct arpreq {
       int32_t ifindex;
       int32_t arp_flags;        /* flags  */
       int32_t arp_hw_addr_len;  /* hardware address length */
       struct  sockaddr arp_pa;  /* protocol address */
       struct  sockaddr arp_ha;  /* hardware address */
       u_char  arp_pad[242];     /* buffer for link specific info. */
};
/*  arp_flags field values */
#define ATF_COM         0x02      /* ARP on ether */
#define ATF_PERM        0x04      /* permanent entry */
#define ATF_PUBL        0x08      /* publish entry */
#define ATF_SNAPFDDI    0x200     /* SNAP - FDDI */
#define ATF_SNAP8025    0x400     /* SNAP - 8025 */
#define ATF_IEEE8025    0x800     /* IEEE - 8025 */
#define ATF_FCSNAP      0x4000    /* Fibre Channel SNAP */
```

The address family for the *arp_pa* **sockaddr** must be **AF_INET**; for the *arp_ha* **sockaddr** it must be **AF_UNSPEC**. The only flag bits that can be written are **ATF_PERM**, and **ATF_PUBL**. Fibre Channel hosts only support the **ATF_PERM** flag. **ATF_PERM** causes the entry to be permanent. **ATF_PUBL** specifies that the ARP code should respond to ARP requests for the indicated host coming from other machines. This allows a host to act as an *ARP server*, which may be useful in convincing an ARP-only machine to talk to a non-ARP machine.

ARP watches passively for hosts impersonating the local host (i.e., a host that responds to an ARP mapping request for the local host's address).

DIAGNOSTICS

duplicate IP address!! sent from ethernet address: %x:%x:%x:%x:%x:%x.

This message printed on the console screen means that ARP has discovered another host on the local network that responds to mapping requests for its own Internet address.

WARNINGS

To enable the **ether** encapsulation method, use the **ifconfig** command (see *ifconfig*(1M)).

AUTHOR

ARP was developed by the University of California, Berkeley.

SEE ALSO

ifconfig(1M), inet(3N), lan(7), arp(1M).

An Ethernet Address Resolution Protocol, RFC826, Dave Plummer, Network Information Center, SRI.

a

NAME

autochanger: schgr, eschgr - SCSI interfaces for medium changer device

DESCRIPTION

An autochanger is a SCSI mass storage device, consisting of a mechanical changer device, one or more data transfer devices (such as optical disk drives), and media (such as optical disks) for data storage. The mechanical changer moves media between storage and usage locations within the autochanger.

Two medium changer drivers (**schgr** or **eschgr**) provide access to the medium changer device; **eschgr** is the current preferred method of access and **schgr** is provided for legacy compatibility. The mechanical changer device can be accessed via these drivers directly to move media within the autochanger.

The **schgr** and **eschgr** medium changer device drivers follow the SCSI specification for medium changer devices to provide a generic medium changer interface, making it feasible to construct an application level driver for any mechanical changer, jukebox, library, or autochanger device (MO, tape, CD-ROM).

Device Naming Convention

The device naming convention for the autochanger driver enables accessing the changer device.

Legacy character device file names reside in **/dev/rac**. Within this directory, names are derived from the **c**#**t**#**d**# device naming convention (explained in *intro*(7)). Unique legacy device names are determined by the card instance, target address of the SCSI changer device and LUN of the SCSI changer device.

Persistent device file names have the form **/dev/rchgr/autochx** for character devices. The card instance, target address and LUN are no longer encoded in the persistent device file name itself (see *intro(7))*.

Major and Minor Number Descriptions

The following shows the bit assignments (**dev_t** format) used by the **schgr** changer driver to access the changer device using legacy device files:

0-7	8-15	16-19	20-22	
MAJOR	INSTANCE	TARGET	LUN	

MAJOR is the major number of the appropriate driver, INSTANCE is the card instance of the SCSI interface to which the changer device is attached, TARGET is the SCSI target address of the changer device, LUN is the SCSI LUN of the changer device.

All fields in the device number are specified in hexadecimal notation. Note that there is no support for hard partitions (sections) in this minor number. If desired, partitioning can be achieved via LVM soft-partitioning schemes.

Note: The major numbers used by the changer drivers are dynamically assigned starting with release HP-UX 11i v3.

Following is a long listing showing the major and minor numbers associated with the device special file name of the changer:

schgr:

crw-rw-rw- 1 root sys 231 0x015000 Apr 22 10:22 /dev/rac/c1t5d0

SCSI MEDIUM CHANGER DEVICE DRIVER

The SCSI medium changer device driver performs moves between different media locations within an autochanger. Each potential media location has a specific element address and is one of the following element types:

storage — A location to hold a unit of media not currently in use. Typically most media will be located in this type of element.

import/export — A location for inserting and removing media from the device. Movement of a unit of media to this type of location is in effect an eject operation. Movement of a unit of media from this type of location is a load operation.

data transfer — A location for accessing media data. This is generally the location of a device that reads and/or writes data on the media being handled by the media changer device. Movement to this type of location is a physical-media-mount operation. Movement from this type of location is a physical-media-unmount operation.

media transport A location for media movement. Media is generally temporarily located in this type of element only during actual media movement.

Changer Control Requests

The following ioctl functions and structure definitions are included from **<sys/scsi.h>**:

```
#define SIOC_INIT_ELEM_STAT    _IO('S', 51)
#define SIOC_ELEMENT_ADDRESSES _IOW('S', 52, struct element_addresses)
#define SIOC_ELEMENT_STATUS    _IOWR('S', 53, struct element_status)
#define SIOC_RESERVE           _IOW('S', 54, struct reservation_parms)
#define SIOC_RELEASE           _IOW('S', 55, struct reservation_parms)
#define SIOC_MOVE_MEDIUM       _IOW('S', 56, struct move_medium_parms)
#define SIOC_EXCHANGE_MEDIUM   _IOW('S', 57, struct exchange_medium_parms)

/* structure for SIOC_ELEMENT_ADDRESSES ioctl */
struct element_addresses {
        unsigned short  first_transport;
        unsigned short  num_transports;
        unsigned short  first_storage;
        unsigned short  num_storages;
        unsigned short  first_import_export;
        unsigned short  num_import_exports;
        unsigned short  first_data_transfer;
        unsigned short  num_data_transfers;
};

/* structure for SIOC_ELEMENT_STATUS ioctl */
struct element_status {
       unsigned short element;         /* element address */

       unsigned int  resv1:2;
       unsigned int  import_enable:1; /* allows media insertion (load) */
       unsigned int  export_enable:1; /* allows media removal (eject) */
       unsigned int  access:1;        /* transport element accessible */
       unsigned int  except:1;        /* is in an abnormal state */
       unsigned int  operatr:1;       /* medium positioned by operator */
       unsigned int  full:1;          /* holds a a unit of media */

       unsigned char resv2;
       unsigned char sense_code;      /* info. about abnormal state */
       unsigned char sense_qualifier; /* info. about abnormal state */

       unsigned int  not_bus:1;       /* transfer device SCSI bus differs */
       unsigned int  resv3:1;
       unsigned int  id_valid:1;      /* bus_address is valid */
       unsigned int  lu_valid:1;      /* lun is valid */
       unsigned int  sublu_valid:1;   /* sub_lun is valid */
       unsigned int  lun:3;           /* transfer device SCSI LUN */

       unsigned char bus_address;     /* transfer device SCSI address */
       unsigned char sub_lun;         /* sub-logical unit number */

       unsigned int  source_valid:1;  /* source_element is valid */
       unsigned int  invert:1;        /* media in element was inverted */
       unsigned int  resv4:6;

       unsigned short source_element;  /* last storage medium location */
       char          pri_vol_tag[36]; /* volume tag (device optional) */
       char          alt_vol_tag[36]; /* volume tag (device optional) */
       unsigned char misc_bytes[168]; /* device specific */
};

/* structure for SIOC_RESERVE and SIOC_RELEASE ioctls */
struct reservation_parms {
```

```
        unsigned short  element;
        unsigned char   identification;
        unsigned char   all_elements;
};

/* structure for SIOC_MOVE_MEDIUM ioctl */
struct move_medium_parms {
        unsigned short  transport;
        unsigned short  source;
        unsigned short  destination;
        unsigned char   invert;
};

/* structure for SIOC_EXCHANGE_MEDIUM ioctl */
struct exchange_medium_parms {
        unsigned short  transport;
        unsigned short  source;
        unsigned short  first_destination;
        unsigned short  second_destination;
        unsigned char   invert_first;
        unsigned char   invert_second;
};
```

SIOC_INIT_ELEM_STAT
Cause the media changer device to take inventory. As a result, the media changer device determines the status of each and every element address, including the presence or absence of a unit of media. This is a mechanical operation which can take time. This function only necessary in the event of a severe error of the media changer.

SIOC_ELEMENT_ADDRESSES
Determine the element addresses supported by a media changer device. The first valid element address and the number of elements is indicated for each element type. These element addresses may be used as source and destination location arguments.

SIOC_ELEMENT_STATUS
Determine the status of an element. The element address for which status information is requested is specified via the **element** field. The resulting status data indicates the presence or absence of a unit of media in that element address as well as other information about the element address.

SIOC_RESERVE and **SIOC_RELEASE**
Control access to element addresses. Depending on the device, reservations may limit operator control of those element addresses in the media changer device. Specific element addresses can be reserved to handle interlocking between multiple requesters if each requester has a unique reservation identification. The value zero in the **all_elements** field specifies that a single element address should be reserved or released. An element address reserved in this manner can not be reserved by another single element address reservation using a different reservation identification. The **reservation** field specifies the reservation identification. The **element** field specifies the element address to be reserved.

The value "1" in the **all_elements** field indicates that all element addresses should be reserved. The **reservation** and **element** fields should contain the value zero since these fields are not meaningful when reserving all element addresses. Reserving all element addresses is primarily useful for limiting operator control.

SIOC_MOVE_MEDIUM and **SIOC_EXCHANGE_MEDIUM**
Reposition unit(s) of media. Depending on the source and destination element types, this may result in a media load, eject, or simple repositioning. Media can be "flipped" using values of "1" in the **invert, invert_first,** or **invert_second** fields. The **SIOC_EXCHANGE_MEDIUM** ioctl repositions two different units of media. One unit of media is moved from the element specified by the **source** field to the element specified by the **first_destination** field. A second unit of media is moved from the element specified by the **first_destination** field to the element specified by the **second_destination** field. In an autochanger with multiple changer mechanisms, or a media staging area, an exchange occurs if the **source** and **second_destination** fields are the same.

DEFAULT CONFIGURATIONS

By default, **schgr** and **eschgr** are not included in the system configuration (**/stand/system**) file.

EXAMPLES

The following example uses the **SIOC_ELEMENT_ADDRESSES** and **SIOC_ELEMENT_STATUS ioctl** functions to get bus address information about the drives in an autochanger device:

```
int                         last_drive_el;
struct element_addresses    el_addrs;
struct element_status       el_stat; drive[1024];
int fd = -1, error = 0, i = 0;


fd = open("/dev/rchgr/autoch0",O_RDWR);
if ((error = ioctl(fd, SIOC_ELEMENT_ADDRESSES, &el_addrs)) != 0) {
   perror("ioctl: SIOC_ELEMENT_ADDRESSES");
   return -1;
} else {
   last_drive_el = el_addrs.first_data_transfer
                 + el_addrs.num_data_transfers - 1;
   for (i = el_addrs.first_data_transfer; i <= last_drive_el; i++) {
      el_stat.element = i;
      if ((error = ioctl(fd, SIOC_ELEMENT_STATUS, &el_stat)) != 0) {
         perror("ioctl: SIOC_ELEMENT_ADDRESSES");
         return -1;
      } else {
         /*
          * You may wish to also check some of the other fields
          * in the el_stat structure to verify that the data is
          * valid.  Fields: el_stat.access (ac accessible),
          * el_stat.except (exception).
          */
         if (! el_stat.not_bus && el_stat.id_valid) {
          drive[i].bus_address = el_stat.bus_address;
            if (! el_stat.lu_valid) {
                drive[i].lun = 0;
            } else {
                drive[i].lun = el_stat.lun;
            }
         }
      }
   }
}
```

WARNINGS

Some non-HP media changer devices do not support the **SIOC_INIT_ELEM_STAT** and **SIOC_ELEMENT_STATUS ioctls**.

Some older media changer devices do not support the **SIOC_EXCHANGE_MEDIUM ioctl**. For these devices, multiple **SIOC_MOVE_MEDIUM** ioctl operations may be used to accomplish the same results, provided a suitable temporary element address may be found.

SEE ALSO

insf(1M), mknod(1M), scsictl(1M), ioctl(2), scsi(7), scsi_ctl(7), intro(7).

NAME

blmode - terminal block mode interface

DESCRIPTION

This terminal interface adds functionality to the current *termio*(7) functionality to allow for efficient emulation of MPE terminal driver functionality. Most importantly, it adds the necessary functionality to support block mode transfers with HP terminals. The block mode interface only affects input processing and does not affect write requests. Write requests are always processed as described in *termio*(7). In character mode the terminal sends each character to the system as it is typed. However, in block mode data is buffered and possibly edited locally in the terminal memory as it is typed, then sent as a block of data when the **Enter** key is pressed on the terminal. During block mode data transmissions, the incoming data is not echoed and no special character processing is performed, other than recognizing a data block terminator character. For subsequent character mode transmissions, the existing termio state continues to determine echo and character processing.

There are two parts of the block mode protocol. The first part is the block mode handshake, which works as follows:

- At the beginning of a read, a *trigger* character is sent to the terminal to notify it that the system is requesting a block of data. (The *trigger* character, if defined, is sent at the beginning of all reads, whether character or block. The *trigger* character must be defined for block mode reads.)
- After receiving the *trigger* character, the terminal waits until the user has typed data into the terminal's memory and pressed the terminal **Enter** key. The terminal then sends an *alert* character to the system to notify it that the terminal has a block of data to send.
- The system may then send user-definable cursor positioning or other data sequences to the terminal. When that is done, the system sends another *trigger* character to the terminal, repeating the cycle.

The second part of the block mode protocol is the block mode transmission. During this transmission of data, the incoming data is not echoed and no special character processing is performed, other than recognizing the data block termination character. It is possible to bypass the block mode handshake and have the block mode transmission occur after the first *trigger* character is sent.

To prevent data loss, XON/XOFF flow control should be used between the system and the terminal. The IXOFF bit should be set and the terminal strapped appropriately. If flow control is not used, it is possible for incoming data to overflow and be lost. (Note: some older terminals do not deal correctly with this flow control.)

It is possible to intermix both character mode and block mode data transmissions. If block mode transmissions are enabled, all transfers are handled as block mode transfers. When block mode transmissions are not enabled, character mode transmissions are processed as described in *termio*(7). If block mode transmissions are not enabled, but an *alert* character is received anywhere in the input data, the transmission mode is switched to block mode automatically for a single transmission.

Read requests that receive data from block mode transmissions will not be returned until the transmission is complete; i.e., the terminal has transmitted all characters. If the read is satisfied by byte count or if a data transmission error occurs, any subsequent data will be discarded. The read waits until completion of the data transmission before returning.

The data block terminator character is included in the data returned to the user, and is included in the byte count. If the number of bytes transferred by the terminal in a block mode transfer exceeds the number of bytes requested by the user, the read returns the requested number of bytes, and the remaining bytes are discarded. The user can determine if data was discarded by checking the last character of the returned data. If the last character is not the terminator character, more data was received than was requested, and data was discarded.

If desired, the application program can provide its own handshake mechanism in response to the *alert* character by selecting the OWNTERM mode. With this mode selected, the driver completes a read request when the *alert* character is received. The second *trigger* is sent by the driver when the application issues the next read.

Several special characters (both input and output) are used with block mode. These characters and the normal values used for block mode are described below. The initial value for these characters is 0377, which causes them to be disabled.

CBTRIG1C (DC1) is the initial *trigger* character sent to the terminal at the beginning of a read request.

CBTRIG2C (DC1) is the secondary *trigger* character sent to the terminal after the *alert* character has been received.

CBALERTC (DC2) is the *alert* character sent by the terminal in response to the first *trigger* character. It signifies that the terminal is ready to send the data block. The *alert* character can be escaped by preceding it with a backslash (****).

CBTERMC (RS) is sent by the terminal after the block mode transfer is complete. It signifies the end of the data block to the computer.

The two *ioctl*(2) requests that apply to block mode use the **blmodeio** structure, which defined in **<blmodeio.h>**, and includes the following members:

```
unsigned long   cb_flags;     /* Modes */
unsigned char   cb_trig1c;    /* First trigger */
unsigned char   cb_trig2c;    /* Second trigger */
unsigned char   cb_alertc;    /* Alert character */
unsigned char   cb_termc;     /* Terminating char */
unsigned char   cb_replen;    /* cb_reply length */
char            cb_reply[];   /* optional reply */
```

The *cb_flags* member controls the basic block mode protocol:

CB_BMTRANS	0000001	Enable mandatory block mode transmission.
CB_OWNTERM	0000002	Enable user control of handshake.

The **CB_BMTRANS** bit is only effective when the **ICANON** flag in *termio*(7) is set. If **ICANON** is clear, all transfers are done in raw mode, regardless of the **CB_BMTRANS** bit. If **CB_BMTRANS** is not set, input processing is performed as described in *termio*(7). During this time, if the *alert* character is defined and is detected anywhere in the input stream, the input buffer is flushed and block-mode handshake is invoked. The system then sends the *cb_trig2c* character to the terminal, and a block mode transfer follows. The *alert* character can be escaped by preceding it with a backslash (****).

If **CB_BMTRANS** is set, then all transmissions are processed as block mode transmissions. Block mode handshake is not required and data read is processed as block mode transfer data. Block mode handshake can still be invoked by receipt of an *alert* character as the first character received. Reads issued while the **CB_BMTRANS** bit is set cause any existing input buffer data to be flushed.

If **CB_OWNTERM** is set, reads are terminated upon receipt of a non-escaped *alert* character. No input buffer flushing is performed and the *alert* character is returned in the data read. This allows application code to perform its own block-mode handshaking. If the bit is clear, an *alert* character causes normal block mode handshaking to be used.

The initial **cb_flags** value is all-bits-cleared.

The **cb_trig1c** character is the initial *trigger* character sent to the terminal at the beginning of a read request. The initial value is undefined (0377); i.e., no *trigger* character is sent.

The **cb_trig2c** character is the secondary *trigger* character sent to the terminal after the *alert* character has been received. The initial value is undefined (0377).

The **cb_alertc** character is the *alert* character sent by the terminal in response to the first *trigger* character sent by the computer. It signifies that the terminal is ready to transmit data. The initial value is undefined (0377).

The **cb_termc** character is sent by the terminal after the block mode transfer has completed. It signifies the end of the data block to the computer. The initial value is undefined (0377).

The **cb_replen** member specifies the length in bytes of the **cb_reply** array. The maximum length of the **cb_reply** array is **NBREPLY** bytes. If set to zero, the *cb_reply* string is not used. It is initially set to zero.

The *cb_reply* array contains a string to be sent out after receipt of the *alert* character but before the second *trigger* character is sent by the computer. Any character can be included in the reply string. The number of characters sent is specified by **cb_replen**. The maximum length of the **cb_reply** array is **NBREPLY** bytes. The initial value of all characters in the **cb_reply** array is null.

On systems that support process group control, *ioctl* requests are restricted from use by background processes, unless otherwise noted for a specific request. An attempt to issue an *ioctl* request from a background process causes the process to block and may cause a **SIGTTOU** signal to be sent to the process group.

The primary *ioctl*(2) calls have the form:

```
int ioctl(int fildes, int request, struct blmodeio *arg);
```

Requests using this form include:

CBGETA Get the parameters associated with the block mode interface and store them in the *blmodeio* structure referenced by *arg*. This request is allowed from a background process. However, the information may be subsequently changed by a foreground process.

CBSETA Set the parameters associated with the block mode interface from the *blmodeio* structure referenced by *arg*. The change is immediate.

RETURN VALUE

Refer to *read*(2), *write*(2), and *ioctl*(2).

ERRORS

If an error value is returned during a read, it is possible for the user's buffer to be altered. In this case, the data in the user's buffer should be ignored because it is incomplete.

The global variable *errno* will be set to indicate the following error, in addition to those errors described on *read*(2), *write*(2), and *ioctl*(2):

[EIO] A read error occurred during the transmission of the block mode data block.

WARNINGS

The [EIO] error that is returned for read errors can be caused by many events. The read returns [EIO] for transmission, framing, parity, break, and overrun errors, or if the internal timer expires. The internal timer starts when the second *trigger* character is sent by the computer, and ends when the terminating character is received by the computer. The length of this timer is determined by the number of bytes requested in the read and the current baud rate, plus an additional ten seconds.

AUTHOR

The *blmode* driver was developed by HP.

SEE ALSO

termio(7).

NAME

cent - Centronics-compatible interface

DESCRIPTION

cent is a simple, widely used communication protocol most commonly associated with printers, plotters and scanners. It is an eight-bit parallel data interface with additional control signals from the host computer, and status signals from the peripheral.

The **cent** interface driver does no character processing; that is, it does not interpret the data being transferred between computer and peripheral. Therefore, all bytes sent to or received from a device are handled without alteration. The **cent** interface driver always operates in **raw mode**; therefore, any desired data interpretation must be performed by a user program (such as the “lp” spooler in conjunction with an appropriate model file). The **cent** driver supports six different handshake modes for data transfer. The last four bits of the minor number of the device special file specify the mode used. The format of the device minor number is:

0x*II000A*

where each letter after the “0x” prefix represents a single hexadecimal digit, as follows:

II Specifies the instance number of the centronic interface.

000 Always zero.

A Specifies the handshake mode. The handshake modes are:

mode 1 Automatic handshaking using both ACK and BUSY.
Minor number format: **0x***II0***001**.

mode 2 Automatic handshaking using only BUSY.
Minor number format: **0x***II0***002**.

mode 3 Bidirectional read/write used for ScanJet.
Minor number format: **0x***II0***003**.

mode 4 Stream mode. Data is essentially transmitted to the peripheral without any handshaking protocol.
Minor number format: **0x***II0***004**.

mode 5 Pulsed mode using both ACK and BUSY for automatic handshaking. Similar to mode 1 except that the data strobe line, **nSTROBE**, is pulsed for a fixed amount of time by the sender, then released.
Minor number format: **0x***II0***005**.

mode 6 Pulsed mode, using only BUSY for automatic handshaking. Similar to mode 1 except that the data strobe line, **nSTROBE**, is pulsed for a fixed amount of time by the sender, then released.
Minor number format: **0x***II0***006**.

Modes 1 and 2 support most HP ∗Jet series printers (LaserJet, DeskJet, QuietJet, etc.).

AUTHOR

cent was developed by HP.

SEE ALSO

lp(1), ioctl(2), intro(7), lp(7).

NAME

clone - opens a major and minor device pair on a STREAMS driver

DESCRIPTION

The **clone** driver is a "pass through" device driver that allows other drivers to select unique minor device numbers on each **open()**. In effect, the driver passes an open operation through to the other driver. This mechanism allows for multiple instantiations of a driver, each with a different minor number, through a single device file.

When the **clone** driver is opened, it is passed a major and minor device number by the operating system. The major number is the **clone** driver's major number (72), and the minor number is the major number of the driver the user wishes to clone (referred to here as the target driver). The **clone** driver calls the open routine of the target driver with the **CLONEOPEN** flag which specifies a clone open. The target driver's open routine allocates an unused minor number. The target driver must use **makedev** to make a new device number for the newly created device, and must set ***devp** to the new device number returned by **makedev**. The new device number is returned to the **clone** open through ***devp**. The **clone** open then returns to the user a file descriptor that points to the new instantiation of the target driver.

The **echo** driver is an example of a clonable driver.

Notes

It is not possible to do multiple opens of a device with the same major and minor number using the **clone** driver. This is because the **clone** driver is only given the major number of the driver to be cloned, and that driver will then select a minor number which has not been opened.

When called with a pathname which corresponds to the clonable driver, **stat()** will return different results than **fstat()** when it is called on a file descriptor returned from **open()** of the same clonable driver pathname.

RETURN VALUES

If the **clone** driver is given an invalid minor number, or if the driver indicated is not a clonable driver, the **open()** fails and **errno** is set to [ENXIO].

SEE ALSO

open(2), fstat(2).

NAME

console, systty, syscon - system console interface

DESCRIPTION

/dev/console provides a **termio** interface to the device configured as the system console. The *init*(1M) manpage discusses the uses of **/dev/systty** and **/dev/syscon**.

Output data normally sent to the console, either through **/dev/console** or generated by a kernel **printf()**, may be redirected to another terminal or pseudo-terminal device through the **TIOCCONS** **ioctl()**. See *termio*(7) for details.

FILES

/dev/console
/dev/systty
/dev/syscon

SEE ALSO

init(1M), termio(7).

STANDARDS CONFORMANCE

console: SVID2, SVID3, XPG2

d

NAME

ddfa - Data Communications and Terminal Controller (DTC) Device File Access (DDFA) software

DESCRIPTION

The Data Communications and Terminal Controller (DTC) Device File Access (DDFA) software allows access from HP-UX system utilities and user applications to terminal servers using standard HP-UX structures. DDFA provides an interface to remote LAN-connected terminal server ports that is similar to the interface for local directly-connected ports.

The basic principle is that a daemon is created for each configured terminal server port based on information in a configuration file (a Dedicated Ports file). When the daemon is spawned, it takes a **pty** from the pool and creates a device file with the same major and minor number as the **pty** slave. The device file is known as the "pseudonym" and utilities and applications use the pseudonym to access the terminal server port by exercising standard HP-UX system functions (**open()**, **close()**, **read()**, **write()**, and **ioctl()**). The daemon listens on the **pty** until an application does an **open()** on the pseudonym. It then sets up and manages the connection to the terminal server port until the application does a **close()** on the pseudonym. The end result is that the terminal server port is addressed via a device file, but the mechanism that makes it happen is transparent to the user. A second configuration file (a port configuration file) contains information to profile the terminal server port.

DDFA consists of the following items:

dp Dedicated Ports file. This text file contains the information that DDFA needs to set up and manage a connection between a pseudonym and a terminal server port.

The **dp** file is parsed by the Dedicated Port Parser (**dpp**) which spawns an Outbound Connection Daemon (**ocd**) for each outbound connection specified in the file. The **dp** file is also used by the HP-UX Telnet daemon (**telnetd**) to identify incoming connections from a DTC and map them to a pseudonym (the Telnet port identification feature).

pcf Port Configuration File. This text file is used by DDFA to profile the terminal server port. The generic name of the template file is **pcf**. A port configuration file is referenced by an entry in the Dedicated Ports file (**dp**).

dpp Dedicated Port Parser. This command parses the Dedicated Ports file (**dp**) and spawns an Outbound Connection Daemon (**ocd**) for each valid entry in the **dp** file. It can be run from the shell or it can be included in a system initialization script to automatically run the DDFA software each time the system is booted.

ocd Outbound Connection Daemon. This daemon manages the connection and data transfer to the remote terminal server port. Normally, it is spawned by the Dedicated Ports Parser (**dpp**), but it can be run directly from the shell.

As it starts, it creates its pseudonym for the connection. As it terminates normally, it removes the pseudonym. If the pseudonym is removed while it is running, **ocd** will terminate with an error condition.

ocdebug Outbound Connection Daemon debug mode. This is a special version of **ocd** that contains debugging code. It must be run from the shell.

CONFIGURATION

There are two basic steps to configuring the DDFA software:

- Enter information in the **dp** file.
- Enter information in the port configuration files.

Configuring the dp File

The **dp** file contains one line for each outbound connection that is to be established and one line for each incoming connection request. A default file **/usr/examples/ddfa/dp** should be copied to a new file and the copy edited as needed. It is recommended that a directory be created to hold the **dp** file and the port configuration files.

Each line of the **dp** file must contain the location of the terminal server port and the location of the pseudonym. In addition, for an outbound connection, the port configuration file must be specified and a logging level may be specified.

Configuring the Port Configuration Files

A port configuration file is used to configure individual terminal server ports. A master port configuration file is **/usr/examples/ddfa/pcf**. In practice, it is renamed for each port that needs different configuration values and the values are altered appropriately for the device attached to the port. It is recommended that a directory be created to hold the port configuration files and the **dp** file.

Each line of a port configuration file must consist of a name of a variable and its value. The variable-value pairs contain information on how to open a connection to a terminal server port, how to close a connection to a terminal server port, and how to manage the data transfer to a terminal server port.

Configuring a System Initialization Script

DDFA can be run at boot time by including a reference to **dpp** in a system initialization script. It is recommended that the **-k** option be used when running **dpp** in this environment.

d

KILLING DAEMONS

Note that **ocd** should be killed using **kill -15**. Do not use **kill -9** for this purpose as it does not remove the device file. **ocd** verifies the validity of an existing pseudonym before trying to use it. **dpp** and **ocd** use data stored in the file **/var/adm/utmp.dfa** to verify whether a process still owns a pseudonym before taking it over. If **ocd** finds an unowned pseudonym, it uses it.

ERROR HANDLING

When **ocd** receives a serious error condition, such as when the LAN goes down, it transmits the error condition to the application by closing the **pty**. Any **open()**, **close()**, **read()**, or **write()** to the pseudonym returns the error condition **0 bytes read**. If the pseudonym is the controlling terminal for the group to which the application belongs, **SIGHUP** is sent to all the processes in the group, including the application.

ioctl() LIMITATIONS

Not all **ioctl()** functionality is available, due to the lack of a protocol that allows the transmission of such commands over the LAN to the remote port.

termio Attribute Limitations

The main restrictions on **termio** attributes (see *termio*(7)) include modem signal control and parity checking. The following are not available:

CBAUD IGNPAR INPCK IXANY IXOFF PARMRK

ioctl() Request Limitations

The following **ioctl()** request limitations apply:

CSTOPB flag	DTC only supports one stop bit.
CSIZE	DTC only supports 8 bits per character. Value cannot be modified.
PARODD flag	DTC offers static configuration to handle even or odd parity. It also handles auto parity detection for even or odd parity.
PARENB flags	Enabling/disabling done via static configuration. No programmatic interface supplied.
INPCK flag	No way to separate input from output parity features.
IGNPAR flag	Cannot be configured on DTC.
PARMRK	Bad characters are forwarded to the system without marking them with OFFH or OH.
CBAUD	Speed is part of static configuration.
IXOFF flag	Flow control is enabled if the DTC static configuration specifies an ASCII access mode. If binary is selected, no flow control is provided.
IXON flags	Pacing of output to a terminal via a programmatic interface is enabled when ASCII mode is selected in static port configuration and disabled when binary mode is selected.
IXANY flag	DTC does not offer the ability to restart output on any character received if XOFF was previously received.

HUPCL flag DDFA does not support the hanging up of modem signals on the last close of the device file. If the modem signals used on the DTC drop, the connection is closed.

CLOCAL flag Not supported.

c_flags **IENQACK** not supported.

OFILL, **OFDEL**, **NLDLY**, **CRDLY**, **TABDLY**, **BSDLY**, **FFDLY** not supported by Telnet port identification software.

BINARY mode flags Part of static configuration is done in DTC Manager by selecting binary mode. If switching is enabled, binary can be selected at user interface level. There is no way to automatically negotiate binary mode when proper termio flags are reset when using **telnetd**. Binary/ASCII switching is possible with DDFA. The DTC cannot support large reads in pure binary mode, so transferred blocks of data should not be more than 256 bytes. If half-duplex with remote acknowledgement is implemented, binary applications can be supported.

d

ioctl() System Call Requests

The following **ioctl()** system call limitations apply:

TCSBRK The ability to send a break without waiting for previous data to be sent is not provided at the system level in **telnetd** or DDFA. Receiving a Telnet break command in the DTC allows it to generate a break on asynchronous ports.

TCFLSH The DTC output queue cannot be flushed.

Hardware handshake request
Not supported on DTC.

TCXONC Local handshake cannot be disabled on DTC.

MCGETA Not supported.

MCSETA, **MCSETAF**, **MCSETAW**
There is no way to separately set modem lines of a DTC port.

MCGETT Modem timers, CD timer, connect timer, and disconnect cannot be configured.

CCITT simple, and direct call-in/call-out modes
DTC cannot handle simple mode because there is programmatic interface for modem signals. Call-in mode cannot be simulated if the port is opened, because modem signals (or the call) must be present within 2 minutes or the connection is cleared.

DACIDY get device adapter info
No way to get device adapter information.

Download **ioctl()** **DACRADDR**, **DACDLADDR**, **DACDLGO**, **DACDLVER**
No programmatic call to download the DTC.

DACHWSTATUS, **DACSELFTEST**, **DACLOADED**, **DACISBROKE** status
No programmatic interface to get such info.

DACLOOPBACK **DACSUBTEST** port test

WARNINGS

In order to ensure that commands (such as *ps*) display the correct device file name (that is, the *pseudonym*), all pseudonyms should be placed into the directory **/dev/telnet**. If pseudonyms are not specified for placement in this directory, the correct display of device file names with many commands is not guaranteed.

In addition, in order to ensure that commands (such as **w**, **passwd**, **finger**, and **wall**) work correctly, each pseudonym must be unique in its first 17 characters (including the directory prefix **/dev/telnet/**). If pseudonyms are not unique in their first 17 characters, the correct functioning of many commands is not guaranteed.

Also, in order to reliably handle timing mark negotiations (and ensure that files printing on a printer attached to a terminal server have been completely flushed to that printer), the following line must be added near the end of each printer interface script for printers attached to a terminal server:

```
stty exta <&1 2>/dev/null
```

The printer interface scripts reside in the directory **/etc/lp/interface**. The line must be added just prior to the final **exit** command in each printer interface script.

If this line is not added as specified, the printing reliability of printers attached to a terminal server is not guaranteed.

FILES

/usr/examples/ddfa/dp
/usr/examples/ddfa/pcf
/usr/sbin/dpp
/usr/sbin/ocd
/usr/sbin/ocdebug
/var/adm/dpp_login.bin
/var/adm/utmp.dfa

SEE ALSO

dpp(1M), ocd(1M), ocdebug(1M), ioctl(2), dp(4), pcf(4), ioctl(5), termio(7).

d

NAME

diag0 - diagnostic interface to HP-PB I/O subsystem

DESCRIPTION

diag0 is a diagnostic pseudo-driver, which provides HP support tools with access to the HP-PB I/O subsystem. This driver is used by hardware monitors and tools within the Support Tools Manager (STM), to interact with peripherals connected to the system via HP-PB. The I/O drivers also send diagnostic events to **diag0** for diagnostic logging by the Support Tools Manager.

Without **diag0**, information that could help prevent a peripheral failure will be lost. In addition, if a failure occurs, HP will not have the tools or data to diagnose the cause of the problem in a timely manner. This may cause increased downtime and possible future failures.

AUTHOR

diag0 was developed by HP.

FILES

/stand/vmunix
/dev/diag/diag0
/dev/diag directory containing diagnostic device files

SEE ALSO

stm(1M) from the Support Tools Manager

NAME

diag1 - diagnostic interface to the PCI I/O subsystem

DESCRIPTION

diag1 is a diagnostic pseudo-driver, which provides support tools with access to the PCI I/O subsystem. This driver is used by tools within the Support Tools Manager (STM) to interact with PCI cards connected to the system. Without **diag1**, support tools for PCI cards will not be able to operate.

WARNINGS

diag1 is not supported for HP-UX 11i Version 1.5.

AUTHOR

diag1 was developed by HP.

FILES

/stand/vmunix

/dev/diag/diag1

/dev/diag directory containing diagnostic device files

SEE ALSO

stm(1M) from the Support Tools Manager.

d

NAME

diag2 - interface for diagnostic logging and interface to processors

DESCRIPTION

diag2 is used by hardware monitors and tools within the Support Tools Manager (STM), to interact with processor hardware via Processor Dependent Code (PDC). Without **diag2**, support tools for processors will not be able to operate.

diag2 is also the key component for the following support features:

I/O error logging
Low priority machine check (LPMC) logging
Memory error logging
Pro-active memory page deallocation.

Without the above, information that could help prevent a system or peripheral failure will be lost. In addition, if a failure occurs, HP will not have the tools or data to diagnose the cause of the problem in a timely manner. This may cause increased downtime and possible future failures.

AUTHOR

diag2 was developed by HP.

FILES

/stand/vmunix
/dev/diag/diag2
/dev/diag2
/dev/diag directory containing diagnostic device files

SEE ALSO

stm(1M) from the Support Tools Manager

NAME

disk - direct disk access

DESCRIPTION

This entry describes the actions of HP-UX disk drivers when referring to a disk as either a block-special or character-special (raw) device.

Device File Naming Conventions

Standard disk device files are named according to the following conventions (see *intro*(7)):

Block-mode Devices	**/dev/disk/disk***N*[**_p***X*]
Character-mode Devices	**/dev/disk/disk***N*[**_p***X*]
Legacy block-mode Devices	**/dev/dsk/c***x***t***y***d***n*[**s***m*]
Legacy character-mode Devices	**/dev/rdsk/c***x***t***y***d***n*[**s***m*]

Legacy device special filenames are those used on HP-UX 11i Version 2 and earlier releases. They can still be used for backward compatibility, but only for part of the configuration within the limits of HP-UX 11i Version 2.

The component parts of the device filename are constructed as follows:

N Required. A decimal number corresponding to the instance number assigned to the direct access device by the operating system.

X Required if **_p** is specified. A decimal number corresponding to a partition number.

c Required. Identifies the following hexadecimal digits as the "Instance" of the interface card.

x Hexadecimal number identifying controlling bus interface, also known as the "Instance" of this interface card. The instance value is displayed in the *ioscan*(1M) output, column "I" for the H/W Type, "INTERFACE".
Required.

t Identifies the following hexadecimal digits as a "drive number" or "target".
Required.

y Hexadecimal number identifying the drive or target number (bus address).
Required.

d Identifies the following hexadecimal digits as a "unit number".
Required.

n Hexadecimal unit number within the device.
Required.

s Optional. Defaults to that corresponding to whole disk. Identifies the following value as a "section number".

m Required if **s** is specified. Defaults to section 0 (zero), whole disk. Drive section number.

Assignment of controller, drive, logical unit and section numbers is described in the system administrator manuals for your system.

Block-special access

Block-special device files access disks via the system's block buffer cache mechanism. Buffering is done in such a way that concurrent access through multiple opens and mounting the same physical device is correctly handled to avoid operation sequencing errors. The block buffer cache permits the system to do physical I/O operations when convenient. This means that physical write operations may occur substantially later in time than their corresponding logical write requests. This also means that physical read operations may occur substantially earlier in time than their corresponding logical read requests.

Block-special files can be read and written without regard to physical disk records. Block-special file **read()** and **write()** calls requiring disk access result in one or more **BLKDEV_IOSIZE** byte (typically 2048 byte) transfers between the disk and the block buffer cache. Applications using the block-special device should ensure that they do not read or write past the end of last **BLKDEV_IOSIZE** sized block in the device file. Because the interface is buffered, accesses past this point behave unpredictably.

Character-special access

Character-special device files access disks without buffering and support the direct transmission of data between the disk and the user's read or write buffer. Disk access through the character special file interface causes all physical I/O operations to be completed before control returns from the call. A single read or write operation up to **MAXPHYS** bytes (typically 64 Kbytes or 256 Kbytes) results in exactly one disk operation. Requests larger than this are broken up automatically by the operating system. Since large I/O operations via character-special files avoid block buffer cache handling and result in fewer disk operations, they are typically more efficient than similar block-special file operations.

There may be implementation-dependent restrictions on the alignment of the user buffer in memory for character special file **read()** and **write()** calls. Also, each read and write operation must begin and end on a logical block boundary and must be a whole number of logical blocks in size. The logical block size is a hardware-dependent value that can be queried with the **DIOC_DESCRIBE_EXT** and **DIOC_DESCRIBE** ioctl calls, which are described below.

In addition to reading and writing data, the character-special file interface can be used to obtain device specific information and to perform special operations. These operations are controlled through use of ioctl calls. Details related to these ioctls are contained in **<sys/diskio.h>**.

The **DIOC_DESCRIBE_EXT** and **DIOC_DESCRIBE** ioctl can be used to obtain device specific identification information. The information returned includes the disk's model identification, the disk interface type, maximum offset address, device type, and the disk's logical block size.

The **DIOC_CAPACITY** ioctl can be used to obtain the capacity of a disk device in **DEV_BSIZE** units. (**DEV_BSIZE** is defined in **<sys/param.h>**).

The **DIOC_EXCLUSIVE** ioctl can be used to obtain and release exclusive access to a disk device. Exclusive access is required for some special operations, such as media reformatting, and may be desirable in other circumstances. The value one specifies that exclusive access is requested. The value zero specifies the exclusive access should be released. Exclusive access causes other open requests to fail. Exclusive access can only be granted when the device is not currently opened in block-mode and there is only one open file table entry for that disk device (the one accessible to the exclusive access requester).

ERRORS

The following errors can be returned by a disk device driver call:

[EACCES] Required permission is denied for the the device or operation.

[EIO] I/O error (e.g., media defect or device communication problem).

[EINVAL] From an **open()** call: the device is not a disk device. For other calls: Invalid request or parameter. Note that for legacy, 32-bit access, this error can result when the size of the device overflows the argument of the **DIOC_DESCRIBE** or **DIOC_CAPACITY** ioctls.

[ENXIO] If resulting from an **open()** call, this indicates there is no device at the specified address. For other calls, this indicates the specified address is out of range or the device can no longer be accessed.

WARNINGS

The interaction of block-special and character-special file access to the same **BLKDEV_IOSIZE**-sized block is not specified, and in general is unpredictable.

On some systems, having both a mounted file system and a block special file open on the same device can cause unpredictable results; this should be avoided if possible. This is because it may be possible for some files to have private buffers in some systems.

Although disk devices have historically had small (typically 512-byte) block sizes, some disk devices (such as optical disks and disk arrays) have relatively large block sizes. Applications using direct raw disk access should use **ioctl()** calls to determine appropriate I/O operation sizes and alignments.

Any disk with removable media (for example, floppy or CD-ROM) containing a mounted file system should not be removed prior to being unmounted. Removal of disk media containing mounted file systems is likely to result in file system errors and system panics.

AUTHOR

disk was developed by HP and AT&T.

SEE ALSO

ioscan(1M), mknod(1M), intro(7).

System Administrator manuals included with your system.

NAME

dlpi - data link provider interface

DESCRIPTION

This manual page gives a brief description on DLPI (the data link provider interface) and how to interface with the set of API's that are provided by DLPI.

HP-UX DLPI serves as a Layer 2 (Data Link Layer) of an OSI architecture. DLPI serves as an interface between LAN device drivers and DLPI users. DLPI is intended for use by experienced network users only.

HP-UX DLPI has two broader sets of interface. The first set of interfaces are provided as per the DLPI 2.0 standard and the second set that are HP extensions to the standard.

HP-UX DLPI also provides interfaces to device drivers to interface with STREAMS modules and DLPI applications.

For STREAMS Modules and DLPI Applications

Hewlett-Packard's implementation of DLPI is a Style 2 service provider. The Style 2 provider requires a DLS user to identify a PPA explicitly, using a special attach service primitive. Refer to the *lan*(7) manual page for more information on PPA.

HP DLPI offers the following services to STREAMS modules and DLPI applications:

- Clone (maximum of 3992) and non-clone (maximum of 100) access.
- Support for Ethernet/IEEE802.3, FDDI and Token Ring interfaces.
- Support for connectionless and connection-mode services (connection-mode services are supported only over IEEE802.3 and Token Ring).
- Supports raw-mode services.
- **`I_STR`** ioctl is supported for doing device-specific control and diagnostic requests.
- Support for third-party device drivers.
- Support for all levels of promiscuous mode.

HP DLPI does not offer the following for STREAMS modules and DLPI applications:

- Quality of Service (QOS) management.
- Connection Management STREAMS: **`DL_SUBS_BIND_REQ`** and **`DL_SUBS_UNBIND_REQ`** over connection-oriented STREAMS.
- Acknowledged connectionless-mode services.

The DLPI requests based on DLPI 2.0 standard are defined in **`<dlpi.h>`**; see *dlpi*(4). HP extensions for DLPI are defined in **`<dlpi_ext.h>`**; see *dlpi_ext*(4).

Device File Format

To access LAN drivers via DLPI interface, DLS users must use the following device files:

Name	Type	Major #	Minor #	Access Type
/dev/dlpi	c	72	0x77	Clone access
/dev/dlpiX	c	119	0xX	Non-Clone access

For Device Drivers

HP-UX DLPI is of non-native design. The drivers and DLPI are not coupled together and exists as individual components on the system. The non-native DLPI supports two kinds of drivers. Tightly coupled and loosely coupled drivers.

DLPI provides interfaces to tightly coupled and loosely coupled drivers. DLPI serves as a sole interface to DLS users for tightly coupled drivers. Whereas, a loosely coupled driver depends on DLPI only to provide information to user-space commands *lanscan*(1M) and *nwmgr*(1M) for display purposes.

The interfaces for device drivers is defined in **`<dlpi_drv.h>`**, see *dlpi_drv*(4).

DLPI provides the following functionality for tightly coupled drivers:

- Infrastructure that allows drivers to communicate with upper layer STREAMS modules or applications.

- Infrastructure for protocol, multicast and promiscuous processing.
- Infrastructure for asynchronous processing of control.
- Inbound frame processing.
- Processing link up and down events.
- Repository for all registered interfaces and associated information.
- Outbound processing before hand off to physical drivers.

DLPI provides its services through three header files that are exported. The header files **<dlpi.h>** and **<dlpi_ext.h>** are for user space applications and kernel level STREAMS modules. The header file **<dlpi_drv.h>** is for physical and logical drivers.

WARNINGS

Various implementations of DLPI exists within HP-UX for special technologies like ATM, Hyper Fabric, etc.; but the DLPI that supports LAN class drivers (tightly coupled) is the one covered by this manual page.

The **lanadmin**, **lanscan**, and **linkloop** commands are deprecated. These commands will be removed in a future HP-UX release. HP recommends the use of replacement command *nwmgr*(1M) to perform all network interface-related tasks.

AUTHOR

dlpi was developed by HP, based on DLPI 2.0 standard.

SEE ALSO

lanscan(1M), nwmgr(1M), dlpi(4), dlpi_drv(4), dlpi_ext(4), lan(7).

DLPI Programmer's Guide, 2003, Hewlett-Packard
Driver Development Guide, Hewlett-Packard
Device Driver Reference, Hewlett-Packard

NAME

framebuf - information for raster frame-buffer devices

SYNOPSIS

#include <sys/framebuf.h>

DESCRIPTION

Frame-buffer devices are raster-based displays. These devices use memory-mapped I/O to obtain much higher performance than possible with tty-based graphic terminals. Frame-buffer devices can be accessed directly using this interface, although access through the graphics libraries is recommended. Direct access to frame-buffer devices entails precise knowledge of the frame-buffer architecture being used. Input cannot be piped into or redirected to frame-buffer devices because they are not serial devices.

Each frame-buffer device is associated with a character special file. Major and minor numbers for frame-buffer devices are implementation-dependent. The minor numbers for these devices denote different frame buffers. Implementation-specific details are discussed in the appropriate systems administrator's manuals.

Communication with a frame-buffer device begins with an **open()** system call. Multiple processes can have the frame-buffer device open concurrently.

close() invalidates the file descriptor associated with the frame-buffer device. After a **close()** system call, any access to the frame-buffer device address range might result in a memory fault and the signal SIGSEGV being sent to the process (see *signal*(2)). A process cannot unmap the frame buffer from its address space after the frame-buffer special file is closed. To unmap a frame buffer, use the **GCUNMAP ioctl()** call (see below).

Once a process acquires a lock for the frame-buffer device, it must unlock it explicitly before calling **close()**; see **GCUNLOCK** below.

read() and **write()** system calls are undefined and always return an error. In this case **errno** is set to [ENODEV].

The **ioctl()** system call is used to control a frame-buffer device. The **select()** system call is used to test the frame-buffer device for exceptional conditions. Interrupts from the graphic hardware are considered exceptional conditions. An exceptional condition is automatically cleared after any process that opens the frame-buffer device is notified of the exception by a **select()** call. A call to **select()** for read or write on the file descriptor associated with the frame-buffer device returns a false condition in the read and write bit masks (see *select*(2)).

A frame-buffer device can be accessed by multiple processes at once. However, each process overwrites the output of the others unless one of the lock mechanisms described here or some other synchronization mechanism is used. The lock mechanisms described here are intended for cooperating processes only.

For all frame buffers, data bytes scan from left to right and from top to bottom. A pixel, which is a visible dot on the screen, is associated with a location in the frame buffer. Each device maps one or more bits in memory to a pixel on the screen, although the bits in the frame buffer might not be continuous. Information describing the frame-buffer structure and attributes is found in the **crt_frame_buffer_t** data structure. The **crt_frame_buffer_t** data structure includes the following fields:

```
int crt_id;                     /*display identifier*/
unsigned int crt_attributes;    /*flags denoting attributes*/
char *crt_frame_base;           /*first byte in frame-buffer memory*/
char *crt_control_base;         /*first byte of the control*/
                                /*registers*/
char *crt_region [ CRT_MAX_REGIONS ];
                                /*other regions associated with the*/
                                /*frame-buffer device*/
```

The following are valid **ioctl()** requests:

GCDESCRIBE Describe the size, characteristics, and mapped regions of the frame buffer. The information is returned to the calling process in a **crt_frame_buffer_t** data structure, and the parameter is defined as **crt_frame_buffer_t *arg;**. Although some structure fields contain addresses of one or more frame-buffer device regions, the values of these fields are not always defined. Only after a successful **GCMAP** command is issued (see below) are the correct addresses returned so the user can access the frame-buffer regions directly using the returned addresses.

GCID Provide a device identification number. The parameter is defined as **int *arg;**. The information returned when using this command is a subset of the information provided by **GCDESCRIBE**, and is provided here for backward compatibility only.

GCON, **GCOFF** Turn graphics on or off. These operations are valid for devices whose CRT_GRAPHICS_ON_OFF bit is set in the **crt_attributes** field of the **crt_frame_buffer_t** data structure returned by the **GCDESCRIBE** command. Otherwise, these commands have no effect.

GCAON, **GCAOFF** Turn alpha on or off. These operations are valid for devices whose CRT_ALPHA_ON_OFF bit is set in the **crt_attributes** field of the **crt_frame_buffer_t** data structure returned by the **GCDESCRIBE** command. Otherwise, these commands have no effect.

GCMAP Make the frame-buffer memory, graphics control, and other device regions accessible to the user process making the call. Only processes that request this can directly access frame-buffer memory and control registers. After a successful **GCMAP** call, the fields **crt_frame_base** and **crt_control_base** in the **crt_frame_buffer_t** data structure (returned by a subsequent **GCDESCRIBE** **ioctl()** call), hold the valid addresses of these two regions of the frame buffer. If, for a specific device, more than two regions are to be mapped to the user's address space, the base addresses of up to CRT_MAX_REGIONS extra device regions will be placed in the array **crt_region** in successive order. Only the regions pertinent to a specific frame buffer are mapped. Irrelevant region fields in the **crt_frame_buffer_t** data structure are set to **0**. Use of the *arg* parameter is implementation dependent (see *DEPENDENCIES* below). The base addresses for frame-buffer regions are always page aligned.

f

GCUNMAP Cause access to the frame-buffer memory, graphics control, and possibly other device regions to be removed from the requesting process. The parameter *arg* is ignored and should be set to **0**. Any attempt to access these memory regions after a successful **GCUNMAP** call results in a memory fault and sends the signal SIGSEGV to the process.

GCLOCK Provide for exclusive use of the frame-buffer device by cooperating processes. The calling process either locks the device and continues or is blocked. Blocking in this case means that the call returns only when the frame buffer is available or when the call is interrupted by a signal. If the call is interrupted, it returns an error and **errno** is set to [EINTR]. Waiting occurs if another process has previously locked this frame buffer using the **GCLOCK** command and has not executed a **GCUNLOCK** command yet. The **GCLOCK** command does not prevent other non-cooperating processes from writing to the frame buffer; thus, **GCLOCK** is an advisory lock only. The parameter *arg* is ignored and should be set to **0**.

This call prevents the Internal Terminal Emulator (ITE) from corrupting the state of the graphics hardware (see *termio*(7)). On some systems, as long as the frame buffer is locked with a **GCLOCK** command, the ITE does not output text to it (see *DEPENDENCIES* below). Any attempt to lock the device more than once by the same process fails, and causes **errno** to be set to [EBUSY].

GCLOCK_NOWAIT Provide for exclusive use of the frame-buffer device by cooperating processes. This request has the same effect on the frame-buffer device as does the **GCLOCK** request. However, this call does not wait for the frame buffer to be released by other processes. If the frame-buffer device is locked, the process is not blocked; instead, the system call returns an error and causes **errno** to be set to [EAGAIN]. The parameter *arg* is ignored and should be set to **0**.

GCLOCK_BLOCKSIG

Provide for exclusive use of the frame-buffer device by cooperating processes while blocking all incoming signals for the calling process that otherwise might have been caught. This call is a superset of the **GCLOCK** call. The parameter *arg* is ignored and should be set to **0**. When the display is acquired for exclusive use (and thus locked), all signals sent to the process that otherwise would have been caught by the process "at the time of the" **GCLOCK** call are withheld (blocked) until **GCUNLOCK** is requested. Any attempt to modify the signal mask of the process (see *sigsetmask*(2)) before a **GCUNLOCK** request is made will not have any effect on these blocked signals.

The signals are not blocked until the lock is actually acquired, and might be received while still awaiting the lock.

The signal SIGTSTP is also blocked whether or not it is being caught. The signals SIGTTIN and SIGTTOU are also blocked on frame-buffer devices where the ITE does not output to the device while it is locked. See *DEPENDENCIES* below.

Except for the three signals mentioned above, this call does not block signals that the process did not expect to catch, nor does it block signals that cannot be caught or ignored. This command does not prevent other non-cooperating processes from writing to the frame buffer.

GCLOCK_BLOCKSIG_NOWAIT
Provide for exclusive use of the frame-buffer device by cooperating processes, while blocking all incoming signals for the calling process that otherwise would have been caught. This request has the same effect on the frame-buffer device as does the **GCLOCK_BLOCKSIG** request. However, this call does not wait for the frame buffer to be released by other processes. If the frame-buffer device is locked, the process is not blocked, but the system call returns an error and causes **errno** to be set to [EAGAIN]. The parameter *arg* is ignored and should be set to **0**.

f

GCUNLOCK Relinquish exclusive use of the frame-buffer device. If the device is locked with a **GCLOCK_BLOCKSIG** or **GCLOCK_BLOCKSIG_NOWAIT** **ioctl()** request, the signal mask of the calling process is restored to its state prior to the locking request.

GCRESET Reset the graphic hardware associated with the frame-buffer device to a defined initial state. The call enables the frame-buffer device to respond to the **ioctl()** requests defined here.

GCDMA_OUTPUT Send DMA output to the frame-buffer device. This system call is used to transfer data from a user's array to a rectangular area of the graphics frame-buffer, or optionally, to the device's graphics control space.

The parameters for the DMA are passed in a **crt_dma_ctrl_t** data structure, which includes the following fields:

```
char *mem_addr;       /* Starting address of data
                      being transferred */
char *fb_addr;        /* Address of framebuffer
                      destination */
int length;           /* Number of bytes to transfer,
                      including those "skipped" */
int linelength;       /* Number of bytes written
                      on each framebuffer row */
int skipcount;        /* Number of source bytes to
                      ignore after each "linelength" */
unsigned int flags;   /* Specified options to the driver */
```

To write to the graphics frame-buffer, set **fb_addr** to the address of the upper-left corner of the rectangle to be drawn. The DMA will write **linelength** bytes on each frame-buffer row, ignore the next **skipcount** bytes of memory data, then resume writing at the same starting position on each succeeding frame-buffer row. This is continued until **length** bytes are either written or ignored.

To write to the graphics control space, set **fb_addr** to the address of the first graphics control register to write. In this case, **linelength** and **skipcount** are ignored.

The **flags** parameter specifies options for the DMA. Currently, there are no supported flags and this parameter should be set to zero, otherwise the system call will fail and **errno** is set to [EINVAL].

The DMA has the same effect on the frame-buffer device as using store instructions to write the data. Thus, various graphics control registers may affect the results of the DMA. It is the responsibility of the user program to perform any necessary set-up of the frame-buffer device so that the DMA has the desired results.

The **skipcount** parameter allows the user to refresh a portion of a window image that the user has stored in memory for those cases where only a portion of the image needs to be refreshed. The window image is then a superset of the rectangle being updated, and might thus have different dimensions. The **skipcount** specifies the portion of the row in the larger window image that is excluded from the rectangle. Thus, **linelength** plus **skipcount** would be the number of bytes in each row of the larger window image array.

If a particular framebuffer device supports this system call, the CRT_DMA_OUTPUT flag in the **crt_attributes** field of the **crt_frame_buffer_t** structure is set. Some framebuffer devices supporting DMA might restrict alignment of the various parameters, and are specified in the *DEPENDENCIES* section below. The kernel ensures that these restrictions are obeyed, and if they are not the system call will fail and set **errno** to [EINVAL].

It is the responsibility of the application to guarantee that the system's physical memory is up-to-date by flushing the processor's data cache. One should use the **GCDMA_DATAFLUSH** ioctl to ensure that the data is consistent before initiating a DMA transfer.

GCDMA_DATAFLUSH
: Flush the specified data from the processor's data cache to the system's main memory. This system call is intended to be used before DMA to ensure that an up-to-date version of the data is transferred to the framebuffer or to control space.

  The parameters for the flush are passed in a **crt_flush_t** data structure, which includes the following fields:

```
char *flush_addr;  /* Starting address of data
                      to be flushed */
int flush_len;     /* Number of bytes to flush */
```

  The kernel ensures that the **flush_len** bytes starting at **flush_addr** are consistent in main memory with respect to the cache.

GCSLOT
: Provide pertinent information about the calling process's participation in the system-wide graphics locking mechanism (see the discussion under **GCLOCK** above). The **GCSLOT** request does not carry out any actual locking functionality. The lock information is returned to the calling process in a **crt_gcslot_t** data structure. The parameter is defined as **crt_gcslot_t *arg;**. The **crt_gcslot_t** data structure is defined in the file **<sys/framebuf.h>**.

GCSTATIC_MAP
: Prevent the Internal Terminal Emulator (ITE) from modifying the device's color map.

GCVARIABLE_MAP
: Allow the Internal Terminal Emulator (ITE) to modify the device's color map.

DEPENDENCIES

When requesting **GCMAP**, the parameter *arg* is ignored and should be set to **0**.

All supported ITEs ignore the frame buffer lock for output.

ERRORS

[EAGAIN] The operation would result in suspension of the calling process, but the request was either **GCLOCK_NOWAIT** or **GCLOCK_BLOCKSIG_NOWAIT**.

[EBUSY] Attempted to lock the device, which is already locked by the same process.

[EINTR] A call to **ioctl()** was interrupted by a signal.

[EINVAL] An invalid **ioctl()** command was made.

[ENODEV] Attempted to use **read()** or **write()** system calls on the device.

[ENOMEM] Sufficient memory for mapping could not be allocated.

[ENOSPC] Required resources for mapping could not be allocated.

[ENXIO] The minor number on the device file refers to a nonexistent device.

[EPERM] Requested **GCUNLOCK ioctl()** command, but the device was locked by a different process.

f

AUTHOR

framebuf was developed by HP.

SEE ALSO

mknod(1M), close(2), ioctl(2), lockf(2), open(2), select(2), signal(2), sigsetmask(2), termio(7).

NAME

gang_sched - Gang Scheduler

DESCRIPTION

The gang scheduler permits a set of MPI (Message Passing Interface) processes, or multiple threads from a single process, to be scheduled concurrently as a group.

Gang scheduling is enabled and disabled by setting the **MP_GANG** environment variable to **ON** or **OFF**.

The gang scheduling feature can significantly improve parallel application performance in loaded timeshare environments that are oversubscribed. Oversubscription occurs when the total number of runnable parallel threads, runnable MPI processes, and other runnable processes exceeds the number of processors in the system.

Gang scheduling also permits low-latency interactions among threads in shared-memory parallel applications.

Only applications using the HP-UX V11.0 MPI or pthread libraries can be gang scheduled. Because HP compiler parallelism is primarily built on the pthread library, programs compiled with HP compilers can benefit from gang scheduling.

INTERFACE

The HP-UX gang scheduler is enabled and disabled using an environment variable. The variable is defined as:

MP_GANG [ON | OFF]

Setting **MP_GANG** to **ON** enables gang scheduling and setting it to **OFF** disables it. If **MP_GANG** is not set, or if it is set to an undefined value, no action is taken.

Gang scheduling is a process attribute that is inherited by child processes created by **fork** (see *fork*(2)). The state of gang scheduling for a process can change only following a call to **exec** (see *exec*(2)).

BEHAVIOR

After the **MP_GANG** environment variable is set to **ON**, any MPI or pthread application to execute and find this variable will enable gang scheduling for that process.

Only the pthread and MPI libraries query the **MP_GANG** variable--the operating system does not.

Gang scheduling is an inherited process attribute. When a process with gang scheduling enabled creates a child process, the following occurs:

- The child process inherits the gang scheduling attribute.
- A new gang is formed for the child process. The child does not become part of its parent's gang.

The gang scheduler is engaged only when a gang consists of multiple threads. For a pthread application, this is when a second thread is created. For an MPI application, it is when a second process is added.

As a process creates threads, the new threads are added to the process's gang if gang scheduling is enabled for the process. However, once the size of a gang equals the number of processors in the system, the following occurs:

- New threads or processes are not added to the gang.
- The gang remains intact and continues to be gang scheduled.
- The spill-over threads are scheduled with the regular timeshare policies.
- If threads in the gang exit (thus making room available), the spill-over threads are not added into the gang. However, newly created threads are added into the gang when room is available.

MPI processes are allocated statically at the beginning of execution. When **MP_GANG** is set to **ON**, all processes in an MPI application are made part of the same gang.

Thread and process priorities for gangs are managed identically to timeshare policy. The timeshare priority scheduler determines when to schedule a gang and adheres to the timeshare policies.

Although it is likely that scheduling a gang will preempt one or more higher priority timeshare threads, over the long run the gang scheduler policy is generally fair. All threads in a gang will have been highest priority by the time a gang is scheduled. Because all threads in a gang must execute concurrently, some threads do not execute when they are highest priority (the threads must wait until all other threads have also been selected, allowing other processes to run first).

Gangs are scheduled for a single time-slice. The time-slice is the same for all threads in the system, whether gang-scheduled or not.

When a single gang executes on a system, the gang's threads are assigned to processors in the system and are not migrated to different processors.

In an oversubscribed system with multiple gangs, all gangs are periodically moved in order to give an equalized percentage of CPU time to each of the different threads. This rebalancing occurs every few seconds.

EXTERNAL INFLUENCES

Environment Variables

The following environment variables affect gang scheduling of processes:

- **MP_GANG** enables (when set to **ON**) and disables (when set to **OFF**) gang scheduling of processes. For details see the INTERFACE section of this man page.
- **MP_NUMBER_OF_THREADS** specifies the number of processors available to execute programs compiled for parallel execution. If not set, the default is the number of processors in the system.

PERFORMANCE

Gang scheduling ensures that all runnable threads and processes in a gang are scheduled simultaneously. This improves the synchronization latency in parallel applications. For instance, threads waiting at a barrier do not have to wait for currently unscheduled threads.

However, applications with lengthy parallel regions and infrequent synchronization may perform best when not gang scheduled. For those applications, some threads can be scheduled even if all threads are not scheduled at once.

A gang-scheduled application's performance can be affected by the number of gang-scheduled applications on a system, and by the number of threads in each. The gang scheduler assigns parallel applications to CPUs using a "best fit" algorithm that attempts to minimize CPU overlap among applications.

On systems with complex workloads including gangs of varying sizes, or odd combinations of sizes, the workload may not optimally match the number of CPUs available. In this situation an application may perform better when not gang scheduled, thus enabling some threads to be scheduled rather than waiting for all threads to be scheduled as a gang.

Scheduling Overhead

Gang scheduling incurs overhead when the scheduler collects a set of threads, assigns a set of processors to the threads, and rendezvous the set of threads and processors to achieve concurrent execution.

On an idle system, the gang scheduling overhead can be seen in the execution time of a single parallel application.

Kernel Blocking of Threads

If a thread from a gang blocks in the kernel, the thread's processor is available to run other non-gang-scheduled threads. When the blocked thread resumes and its gang is currently running, the thread can join the other ganged threads without having to rendezvous again.

In a multi-gang environment, thread blocking can result in lower throughput. This occurs if an application's threads block often in the kernel for long periods of time.

Preempting by Realtime Threads

Gang-scheduled threads can be preempted from execution by realtime threads. This affects only the gang-scheduled thread running on the processor being preempted by a realtime thread. The remaining threads of the gang continue to run through the end of their time-slice.

RESTRICTIONS

For this implementation of gang scheduling, the following restrictions exist. Some of these may be removed in future releases.

- Gang scheduling of processes being debugged is not supported. When a debugger attaches to a process, gang scheduling for the process is disabled. This avoids gang scheduling processes with one or more threads stopped by a debugger.

- Gang scheduling is completely shut down when Process Resource Manager (PRM) is enabled.
- If a gang-scheduled process is selected to be swapped out, the process will not be gang-scheduled when it is swapped back in.
- Realtime processes are not gang-scheduled.
- Gang scheduling is only supported for processes with timeshare scheduling policies.
- When a gang-scheduled process contains the maximum number of threads (or the maximum number of processes, for MPI applications), threads or processes created after this point are not scheduled as part of the gang. For details see the BEHAVIOR section of this man page.
- Multiprocess applications that do not use MPI are not supported by the gang scheduler.
- Gang scheduling is not supported for **PTHREAD_SCOPE_PROCESS** threads. From release 11i Version 1.6 of HP-UX, the default scheduling contention scope for threads is **PTHREAD_SCOPE_PROCESS**. If any **PTHREAD_SCOPE_PROCESS** threads are created by an application, the initial thread will be treated as a **PTHREAD_SCOPE_PROCESS**.

g

FILES

The following are libraries used in providing gang scheduling:

/usr/lib/libpthread.1 The pthread library.

/opt/mpi The directory containing MPI libraries and MPI software. HP MPI is an optional product.

SEE ALSO

fork(2), exec(2).

h

NAME

hil - HP-HIL device driver

SYNOPSIS

#include <sys/hilioctl.h>

DESCRIPTION

HP-HIL, the Hewlett-Packard Human Interface Link, is the Hewlett-Packard standard for interfacing a personal computer, terminal, or workstation to its input devices. **hil** supports devices such as keyboards, mice, control knobs, ID modules, button boxes, digitizers, quadrature devices, bar code readers, and touchscreens.

On systems with a single link, HP-HIL device file names use the following format:

/dev/hil*n*

where *n* represents a single digit that specifies the physical HP-HIL device address, which ranges from 1 to 7. For example, **/dev/hil3** is used to access the third HP-HIL device.

On systems with more than one link, HP-HIL device file names use the following format:

/dev/hil_*m.n*

where *m* represents the instance number, and *n* represents the physical HP-HIL device address. For example, **/dev/hil_0.2** would be used to access the second device on the link which has an instance number of zero. Likewise, **/dev/hil_12.7** references the seventh device on the link with instance number twelve.

Note that HP-HIL device addresses are determined only by the order in which devices are attached to the link. The first device attached to the link becomes device one, the second device attached becomes device two, etc.

HP-HIL devices are classified as "slow" devices. This means that system calls to **hil** can be interrupted by caught signals (see *signal*(5)).

hil can only read HP-HIL keyboards in raw keycode mode. Raw keycode mode means that all keyboard input is read unfiltered. HP-HIL keyboards return keycodes that represent key press and key release events.

Use *hilkbd*(7) to read mapped keycodes from HP-HIL keyboards. Use the Internal Terminal Emulator (ITE) described in *termio*(7) to read ASCII characters from HP-HIL keyboards.

System Calls

open(2) gives exclusive access to the specified HP-HIL device. Any previously queued input from the device is discarded. If the device is a keyboard, it is opened in raw keycode mode. A side effect of opening a keyboard in raw keycode mode is that the ITE (see *termio*(7)) and mapped keyboard driver (see *hilkbd*(7)) lose input from that keyboard until it is closed. Only device implemented auto-repeat functionality is available while in raw keycode mode (see HILER1 and HILER2).

The file status flag, O_NDELAY, can be set to enable nonblocking reads (see *open*(2)).

close(2) returns an HP-HIL keyboard to mapped keycode mode, making its input available to the ITE or mapped keyboard driver (see *hilkbd*(7)).

read(2) returns data from the specified HP-HIL device, in time-stamped packets:

```
unsigned char packet_length;
unsigned char time_stamp[4];
unsigned char poll_record_header;
unsigned char data[ packet_length - 6 ];
```

packet_length specifies the number of bytes in the packet including itself, and can range from six to twenty bytes. *time_stamp*, when repacked into an integer, specifies the time, in tens of milliseconds, that the system has been running since the last system boot. The most significant byte of the time stamp is *time_stamp*[0]. *poll_record_header* indicates the type and quantity of information to follow, and reports simple device status information. The number of data bytes is device dependent. Refer to the text listed in *SEE ALSO* for descriptions of the *poll_record_header* and device-specific data.

Usually two system calls are required to read each data packet, the first system call reads the data packet length; the second system call reads the actual data packet. Some devices always return the same amount of data in each packet, in which case the count and the packet can both be read in the same system call.

If the file status flag, O_NDELAY, is set and no data is available, *read*(2) returns **0** instead of blocking.

write(2) is not supported by **hil**.

select(2) can be used to poll for available input from HP-HIL devices. *select*(2) for write or for exception conditions always returns a false indication in the file descriptor bit masks.

ioctl(2) is used to perform special operations on HP-HIL devices. *ioctl*(2) system calls all have the form:

```
int ioctl(int fildes, int request, char *arg);
```

The following *request* codes are defined in <**sys/hilioctl.h**>:

HILID Identify and Describe

This request returns the Identify and Describe Record in the **char** variable to which *arg* points, as supplied by the specified HP-HIL device. The Identify and Describe Record is used to determine the type and characteristics of each device connected to the link. The Identify and Describe Record can vary in length from 2 to 11 bytes. The record contains at least:

- A Device ID byte, and
- A Describe Record Header byte.

The Device ID byte is used to identify the general class of a device, and its nationality in the case of a keyboard or keypad. The Describe Record Header byte describes the position report capabilities of the device. The Describe Record Header byte also indicates if an I/O Descriptor byte follows at the end of the Describe Record. It also indicates support of the Extended Describe and the Report Security Code requests. If the device is capable of reporting any coordinates, the Describe Record contains the device resolution immediately after the Describe Record Header byte. If the device reports absolute coordinates, the maximum count for each axis is specified after the device resolution. The I/O Descriptor byte indicates how many buttons the device has. The I/O Descriptor byte also indicates device proximity detection capabilities and specifies Prompt/Acknowledge functions. All HP-HIL devices support the Identify and Describe request.

HILPST Perform Self Test

This request causes the addressed device to perform its self test, and returns the one-byte test result in the **char** variable to which *arg* points. A test result of zero indicates a successful test, non-zero results indicate device-specific failures. All HP-HIL devices support the Self Test request.

HILRR Read Register

The Read Register request expects an HP-HIL device register address in the **char** variable to which *arg* points, and returns the one-byte contents of that register in ***arg**. The Extended Describe Record indicates whether a device supports the Read Register request.

HILWR Write Register

The Write Register request expects ***arg** to contain a record containing one or more packets of data, each containing the HP-HIL device register address and one or more data bytes to be written to that register. There are two types of Register Writes. Type 1 can be used to write a single byte to each individual device register. Type 2 can be used to write several bytes to one register. The Extended Describe Record indicates if a device supports either or both types of register write requests.

HILRN Report Name

The Report Name request returns the device description string in the character array to which *arg* points. The string may be up to fifteen characters long. The Extended Describe Record indicates support of the Report Name request.

HILRS Report Status

The Report Status request returns the device-specific status information string in the character array to which *arg* points. The string can be up to fifteen bytes long. The Extended Describe record indicates support of the Report Status request.

h

HILED Extended Describe

The Extended Describe request returns the Extended Describe Record in the character array to which *arg* points. The Extended Describe Record may contain up to fifteen bytes of additional device information. The first byte is the Extended Describe Header, which indicates whether a device supports the Report Status, Report Name, Read Register, or Write Register requests. If the device implements the Read Register request, the maximum readable register is specified. If the device supports the Write Register request, the Extended Describe Record specifies whether the device implements either or both of the two types of register writes and the maximum writeable register. If the device supports Type 2 register writes, the maximum write buffer size is specified. The Extended Describe Record can also contain the localization (language) code for a device. Support of the Extended Describe request is indicated in the Describe Record Header byte.

HILSC Report Security Code

The Report Security Code request returns the Security Code Record in the character array to which *arg* points. The Security Code Record can be between one and fifteen bytes of data that uniquely identifies that particular device. Applications can use this request to implement a hardware "key" that restricts each copy of the application to a single machine or user. An application can read the Security Code Record from an HP-HIL ID Module and then verify that the application is running on a specific machine or that the application is being used by a legitimate user. Devices indicate support of the Report Security Code request in the Describe Record Header.

HILER1 Enable Auto Repeat Rate = 1/30 Second

This request is used to enable the "repeating keys" feature implemented by the firmware of some HP-HIL keyboard and keypad devices. It also sets the cursor key repeat rate to 1/30 sec. This request does not use *arg*.

HILER2 Enable Auto Repeat Rate = 1/60 Second

This request is used to enable the "repeating keys" feature implemented in the firmware of some HP-HIL keyboard and keypad devices. It also sets the cursor key repeat rate to 1/60 sec. This request does not use *arg*.

HILDKR Disable Keyswitch Auto Repeat

This request turns off the "repeating keys" feature implemented in the firmware of some HP-HIL keyboard and keypad devices. This request does not use *arg*.

HILP1..HILP7 Prompt 1 through Prompt 7

These seven requests are supported by some HP-HIL devices to give an audio or visual response to the user, perhaps indicating that the system is ready for some type of input. A device specifies acceptance of these requests in the I/O Descriptor Byte in the Describe Record. These requests do not use *arg*.

HILP Prompt (General Purpose)

This request is intended as a general purpose stimulus to the user. Devices accepting this request indicate so in the I/O Descriptor Byte in the Describe Record. This request does not use *arg*.

HILA1..HILA7 Acknowledge 1 through Acknowledge 7

These seven requests are intended to provide an audio or visual response to the user, generally to acknowledge a user's input. The I/O Descriptor Byte in the Describe Record indicates whether an HP-HIL device implements this request. These requests do not use *arg*.

HILA Acknowledge (General Purpose)

The Acknowledge request is intended to provide an audio or visual response to the user. Devices accepting this request indicate so in the I/O Descriptor Byte in the Describe Record. This request does not use *arg*.

ERRORS

[EBUSY] The specified HP-HIL device is already opened.

[EFAULT]	A bad address was detected while attempting to use an argument to a system call.
[EINTR]	A signal interrupted an *open*(2), *read*(2), or *ioctl*(2) system call.
[EINVAL]	An invalid parameter was detected by *ioctl*(2).
[ENXIO]	No device is present at the specified address; see the *WARNINGS* section.
[EIO]	A hardware or software error occurred while executing an *ioctl*(2) system call.
[ENODEV]	*write*(2) is not implemented for HP-HIL devices.

WARNINGS

An [ENXIO] error is returned by *open*(2) and *ioctl*(2) if any attempt is made to access a device while **hil** is reconfiguring the link during power-failure recovery.

hil cannot detect whether or not a device executed an *ioctl*(2) request.

HP-HIL devices have no status bit available to indicate whether they support the HILER1, HILER2, or HILDKR requests.

AUTHOR

hil was developed by HP.

FILES

/dev/hil[1-7]
/dev/hil_*.[1-7]

SEE ALSO

close(2), errno(2), fcntl(2), ioctl(2), open(2), read(2), select(2), signal(5), hilkbd(7), termio(7).

For detailed information about HP-HIL hardware and software in general, see the *HP-HIL Technical Reference Manual*.

h

NAME

hilkbd - HP-HIL mapped keyboard driver

DESCRIPTION

HP-HIL, the Hewlett-Packard Human Interface Link, is the Hewlett-Packard standard for interfacing a personal computer, terminal, or workstation to its input devices. **hilkbd** supplies input from all mapped keyboards on a specified HP-HIL link.

hilkbd returns mapped keycodes, not ASCII characters. "Raw" keycodes are the individual key downstrokes and upstrokes, and are different for each type of keyboard. **hilkbd** maps the raw input into the keycodes and protocol expected by the HP-UX, Pascal Workstation, and BASIC/UX operating systems. The **hil** driver can usurp a keyboard from **hilkbd** by changing it from mapped mode to raw mode.

System Calls

open() gives exclusive access to the keyboard. If there is an ITE (internal terminal emulator) associated with the keyboard, the ITE loses input from the keyboard until the keyboard device is closed. Any previous queued input for the keyboard device is flushed from the input queue.

close() returns control of the keyboard to the ITE, if present. Any unread input is discarded at that time.

h

read() returns data from the keyboard in time-stamped packets:

unsigned char *time_stamp* **[4];**
unsigned char *status* **;**
unsigned char *data* **;**

time_stamp , when repacked into an integer data type of four or more bytes, specifies the time since an arbitrary point in the past (for example, system start-up time). This point does not change between packets, but time during a power failure may or may not be counted. The time is in units of tens of milliseconds.

The *status* byte encodes the state of the keyboard **Shift** and **Ctrl** keys:

0x8X shift and control
0x9X control only
0xAX shift only
0xBX no shift or control

The *data* byte contains the actual keystroke.

If the file status flag O_NDELAY is set, **read()** returns **0** instead of blocking, when no data is available. The **read()** system call on an HP-HIL keyboard is considered "slow"; that is, it can be interrupted by caught signals (see *signal*(2)).

write() is not supported by **hilkbd**.

select() can be used to poll for input to read from **hilkbd** devices. **select()** for write or for exceptional conditions always returns a false indication in the bit masks.

ioctl() is used to perform special operations on the device. **ioctl()** system calls have the form:

int ioctl(int fildes, int request, char *∗***arg);**

The following **hilkbd** request codes are defined in **<sys/hilioctl.h>**:

KBD_READ_CONFIG
Read the configuration code.

This request returns a one-byte configuration code in the **char** variable to which *arg* points. This contains a field, defined by **KBD_IDCODE_MASK**, which specifies the keyboard identification code. The possible values of this field are defined in the header file, and this identification code affects interpretation of the language code. All other fields in the configuration code are currently undefined.

KBD_READ_LANGUAGE
Read the language code.

This request returns a one-byte language code, as read from the keyboard, in the **char** variable to which *arg* points. If there is more than one keyboard, the language is taken from the first keyboard on the link. Interpretation of the language code is affected by the keyboard identification field within the configuration code.

KBD_STATUS Read the keyboard status register.

This request returns a one-byte value containing bit flags specifying the state of the shift and control keys in the **char** variable to which *arg* points:

KBD_STAT_LEFTSHIFT	The left shift key is up
KBD_STAT_RIGHTSHIFT	The right shift key is up
KBD_STAT_SHIFT	Both shift keys are up
KBD_STAT_CTRL	The control key is up

Other bits are undefined.

KBD_REPEAT_RATE
Set the keyboard auto-repeat rate.

The one-byte value to which *arg* points is the negative of the repeat period, in tens of milliseconds. The repeat rate is the reciprocal of the repeat period. A parameter of zero disables auto-repeat.

KBD_REPEAT_DELAY
Set the keyboard auto-repeat delay.

The one-byte value to which *arg* points is the negative of the repeat delay, in tens of milliseconds.

KBD_BEEP Cause an audible beep.

The one-byte value to which *arg* points specifies the volume of the beep, within the range **0** through **KBD_MAXVOLUME**. Implementations with fewer than **KBD_MAXVOLUME** discrete levels of volume will scale the parameter into the smaller range.

ERRORS

[EINVAL] An invalid parameter was detected by **ioctl()**.

[EINTR] A signal was caught during a **read()** system call.

[ENXIO] No keyboard is present on the HP-HIL link specified by the minor number.

[ENODEV] An attempt was made to use **write()** using **hilkbd**.

[EBUSY] The device is already open.

AUTHOR

hilkbd was developed by the Hewlett-Packard Company.

FILES

/dev/hilkbd*

SEE ALSO

mknod(1M), select(2), signal(2), hil(7), termio(7).

NAME

inet - Internet protocol family

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>
```

DESCRIPTION

The internet protocol family is a collection of protocols layered on top of the **Internet Protocol** (IP) network layer, which utilizes the internet address format. The internet family supports the SOCK_STREAM and SOCK_DGRAM socket types.

Addressing

Internet addresses are four byte entities. The include file <**netinet/in.h**> defines this address as the structure **struct in_addr**.

Sockets bound to the internet protocol family utilize an addressing structure called **struct sockaddr_in**. Pointers to this structure can be used in system calls wherever they ask for a pointer to a **struct sockaddr**.

There are three fields of interest within this structure. The first is **sin_family**, which must be set to AF_INET. The next is **sin_port**, which specifies the port number to be used on the desired host. The third is **sin_addr**, which is of type **struct in_addr**, and specifies the address of the desired host.

Protocols

The internet protocol family is comprised of the IP network protocol, Internet Control Message Protocol (ICMP), Transmission Control Protocol (TCP), and User Datagram Protocol (UDP). TCP is used to support the **SOCK_STREAM** socket type while UDP is used to support the **SOCK_DGRAM** socket type. The ICMP message protocol and IP network protocol are not directly accessible.

The local port address is selected from independent domains for TCP and UDP sockets. This means that creating a TCP socket and binding it to local port number 10000, for example, does not interfere with creating a UDP socket and also binding it to local port number 10000 at the same time.

Port numbers in the range 1-1023 inclusive are reserved for use by the super-user only. Attempts to bind to port numbers in this range by non-super-users fail and result in an error returned.

AUTHOR

inet was developed by the University of California, Berkeley.

SEE ALSO

TCP(7P), UDP(7P).

(OBSOLETED)

NAME

iomap - physical I/O address mapping

SYNOPSIS

#include <sys/iomap.h>

DESCRIPTION

The **iomap** mechanism allows the mapping (thus direct access) of physical I/O addresses into the user process address space. For PA-RISC machines, the physical I/O address space begins at **0xf0000000** and extends to **0xffffffff**.

The special (device) files for **iomap** devices are character special files using the dynamic major number allocation scheme.

The minor number for **iomap** devices is of the form:

0xAAAASM

The physical I/O address is formed by prefixing 0xAAAA with 0xF, and by appending 0x000 (this forces the I/O address to be page-aligned). The size of the region to be mapped is given by the expression M*(2^S) 4K pages. For example, the minor number for a device starting at **0xf4000000** that occupies 64MB is **0x4000e1**.

The **iomap** driver must be explicitly added to the **/stand/system** file, the kernel rebuilt, and the system subsequently rebooted prior to first using **iomap**.

I/O space is always mapped with both read and write access rights, regardless of the actual permissions on the device special file.

Multiple processes can have concurrently a single **iomap** device opened and mapped. It is the responsibility of the processes to synchronize their access.

Successive calls to **iomap** to map the same I/O space must be identical to the first mapping. Identical mappings have the same address and size.

Note that a process can additionally share I/O space (mapped by **iomap**) with a kernel driver. However, this is only possible if the driver maps in the I/O space with user read/write access rights using the appropriate driver I/O mapping services. Any I/O space mapped by drivers with kernel read/write access rights cannot be concurrently mapped by processes using **iomap**.

No **read()** or **write()** system calls are supported by the **iomap** driver.

The **ioctl()** function is used to control the **iomap** device. The following **ioctl()** requests are defined in **<iomap.h>**:

IOMAPMAP Map the **iomap** device into user address space at the location specified by the pointer to which the **(void **)** third argument to **ioctl()** points. If the argument points to a variable containing a null pointer, the system selects an appropriate address. **ioctl()** then returns the user address where the device was mapped, storing it at the address pointed to by the third argument (see *EXAMPLES* below). Multiple processes can concurrently have the same **iomap** device mapped.

IOMAPUNMAP Unmap the **iomap** device from the user address space.

close() shuts down the file descriptor associated with the **iomap** device. If the close is for the last system wide open on the device, the **iomap** device is also unmapped from the user address space; otherwise it is left mapped into the user address space (see **IOMAPUNMAP** above).

WARNINGS

Be extremely careful when creating and using **iomap** devices. Inappropriate accesses to I/O devices or RAM can result in a system crash.

ERRORS

[EINVAL] The address field was out of range, or the **ioctl** request was invalid.

[ENOMEM] Not enough memory could be allocated for the mapping.

[EBUSY] Device was already mapped and this mapping was not identical to the initial mapping (same address, size and access rights).

[ENODEV] Read and write calls are unsupported.

[ENXIO] No such device at the address specified by the minor number.

[ENOSPC] Required resources for mapping could not be allocated.

[ENOTTY] Inappropriate **ioctl** request for this device type; *fildes* is not a file descriptor for an **iomap** device file.

EXAMPLES

Consider the following code fragment:

```
#include <sys/iomap.h>
    ...
int fildes;
void *addr;
    ...
    addr = REQUESTED_ADDRESS;
    (void) ioctl(fildes, IOMAPMAP, &addr);
    (void) printf("actual address = 0x%x\n", addr);
```

where **fildes** is an open file descriptor for the device special file and **REQUESTED_ADDRESS** is the address originally requested by the program.

If **addr** is a null pointer, the system selects a suitable address then returns the selected address in *addr*.

If the value in *addr* is not a null pointer, it is used as a specified address for allocating memory. If the specified address cannot be used, an error is returned (see *ERRORS*).

SEE ALSO

mknod(1M).

NAME

IP - Internet Protocol

SYNOPSIS

```
#include <sys/socket.h>
#include <netinet/in.h>

s = socket(AF_INET, SOCK_DGRAM, 0);
```

DESCRIPTION

IP is the network-layer protocol used by the Internet protocol family. It encapsulates TCP and UDP messages into datagrams to be transmitted by the network interface. Normally, applications do not need to interface directly to IP. However, certain multicast socket options are controlled by passing options to the **IPPROTO_IP** protocol level through a UDP socket, and IP Type of Service is controlled by passing an option to the **IPPROTO_IP** protocol level through either a TCP or UDP socket. (See the *getsockopt*(2) manual page.)

The following socket options are defined in the include file **<netinet/in.h>**. The type of the variable pointed to by the *optval* parameter is indicated in parentheses. The data types **struct ip_mreq** and **struct in_addr** are defined in **<netinet/in.h>**.

IP_TOS
: (**unsigned int**) Sets the IP Type of Service. Allowable values for optval are 4 for high reliability, 8 for high throughput, and 16 for low delay. Other values will not return an error, but may have unpredictable results. Default: zero.

IP_ADD_MEMBERSHIP
: (**struct ip_mreq**) Requests that the system join a multicast group.

IP_DROP_MEMBERSHIP
: (**struct ip_mreq**) Allows the system to leave a multicast group.

IP_MULTICAST_IF
: (**struct in_addr**) Specifies a network interface other than the default to be used when sending multicast datagrams through this socket. Default: multicast datagrams are sent from the interface associated with the specific multicast group, with the default multicast route or with the default route.

IP_MULTICAST_LOOP
: (**unsigned char**; boolean) Enables or disables loopback in the IP layer for multicast datagrams sent through this socket. The value of the variable pointed to by *optval* is zero (disable) or non-zero (enable). This option is provided for compatibility only. Normally, multicast datagrams are always looped back if the system has joined the group. See *DEPENDENCIES* below. Default: enabled.

IP_MULTICAST_TTL
: (**unsigned char**) Specifies the time-to-live value for multicast datagrams sent through this socket. The value of the variable pointed to by *optval* can be zero through 255. Default: one.

IP_ADD_MEMBERSHIP requests that the system join a multicast group on the specified interface. For example:

```
struct ip_mreq mreq;
mreq.imr_multiaddr.s_addr = net_addr("224.1.2.3");
mreq.imr_interface.s_addr = INADDR_ANY;
setsockopt(s, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mreq, sizeof(mreq));
```

A system must join a group on an interface in order to receive multicast datagrams sent on the network to which that interface connects. If **imr_interface** is set to **INADDR_ANY**, the system joins the specified group on the interface that datagrams for that group would be sent from, based the routing configuration. Otherwise, **imr_interface** should be the IP address of a local interface. An application can join up to **IP_MAX_MEMBERSHIPS** multicast groups on each socket. **IP_MAX_MEMBERSHIPS** is defined in **<netinet/in.h>**. However, each network interface may impose a smaller system-wide limit because of interface resource limitations and because the system uses some link-layer multicast addresses.

The application must also bind to the destination port number in order to receive datagrams that are sent to that port number. If the application binds to the address **INADDR_ANY**, it may receive all datagrams that are sent to the port number. If the application binds to a multicast group address, it may receive only datagrams sent to that group and port number. It is not necessary to join a multicast group in order to send datagrams to it.

IP_DROP_MEMBERSHIP allows the system to leave a multicast group. For example:

```
struct ip_mreq mreq;
mreq.imr_multiaddr.s_addr = net_addr("224.1.2.3");
mreq.imr_interface.s_addr = INADDR_ANY;
setsockopt(s, IPPROTO_IP, IP_DROP_MEMBERSHIP, &mreq, sizeof(mreq));
```

The system remains a member of the multicast group until the last socket that joined the group is closed or has dropped membership in the group.

IP_MULTICAST_IF specifies a local network interface to be used when sending multicast datagrams through this socket. For example:

```
#include <arpa/inet.h>
struct in_addr addr;
addr.s_addr = inet_addr("192.1.2.3");
setsockopt(s, IPPROTO_IP, IP_MULTICAST_IF, &addr, sizeof(addr));
```

Normally, applications do not need to specify the interface. By default, multicast datagrams are sent from the interface specified by the routing configuration, namely the interface associated with the specific multicast group, with the default multicast route or with the default route. If **addr** is set to the address **INADDR_ANY**, the default interface is selected. Otherwise, **addr** should be the IP address of a local interface.

IP_MULTICAST_LOOP enables or disables loopback for multicast datagrams sent through this socket. For example:

```
unsigned char loop = 1;
setsockopt(s, IPPROTO_IP, IP_MULTICAST_LOOP, &loop, sizeof(loop));
```

Note that the type of the *optval* parameter is **unsigned char** instead of **int**, which is common for boolean socket options. This option is provided for compatibility only. Normally, if a multicast datagram is sent to a group that the system has joined, a copy of the datagram is always looped back and delivered to any applications that are bound to the destination port. See *DEPENDENCIES* below.

IP_MULTICAST_TTL controls the scope a multicast by setting the time-to-live value for multicast datagrams sent through this socket. For example:

```
unsigned char ttl = 64;
setsockopt(s, IPPROTO_IP, IP_MULTICAST_TTL, &ttl, sizeof(ttl));
```

Note that the type of *optval* parameter is **unsigned char** instead **int**, which is common for socket options. By default, the time-to-live field (TTL) is one, which limits the multicast to the local network. If the TTL is zero, the multicast is limited to the local system (loopback). If the TTL is two, the multicast can be forwarded through at most one gateway; and so forth. Multicast datagrams can be forwarded to other networks only if there are special multicast routers on the local and intermediate networks.

DEPENDENCIES

The behavior of **IP_MULTICAST_LOOP** depends on the network driver and interface card. Normally, loopback cannot be disabled, even if **IP_MULTICAST_LOOP** is set to zero, because it occurs in the driver or in the network interface. However, if the outbound interface is **lo0** (127.0.0.1), or if **IP_MULTICAST_TTL** is set to zero, setting **IP_MULTICAST_LOOP** to zero will disable loopback for multicast datagrams sent through the socket.

ERRORS

One of the following errors may be returned if a call to **setsockopt()** or **getsockopt()** fails.

[EADDRINUSE]	The specified multicast group has been joined already on socket.
[EADDRNOTAVAIL]	The specified IP address is not a local interface address; or there is no route for the specified multicast address; or the specified multicast group has not been joined.
[EINVAL]	The parameter *level* is not **IPPROTO_IP**; or *optval* is the NULL address; or the specified multicast address is not valid.
[ENOBUFS]	Insufficient memory is available for internal system data structures.
[ENOPROTOOPT]	The parameter *optname* is not a valid socket option for the **IPPROTO_IP** level.

[EOPNOTSUPP]	The socket type is not **SOCK_DGRAM**.
[ETOOMANYREFS]	An attempt to join more than **IP_MAX_MEMBERSHIPS** multicast groups on a socket.

AUTHOR

The socket interfaces to IP were developed by the University of California, Berkeley. Multicast extensions were developed by the Stanford University.

SEE ALSO

bind(2), getsockopt(2), recv(2), send(2), socket(2), inet(7F).

NAME

IPv6, ipv6, ip6 - Internet Protocol Version 6

SYNOPSIS

```
#include <sys/socket.h>
#include <netinet/in.h>

s = socket(AF_INET6, SOCK_DGRAM, 0);
s = socket(AF_INET6, SOCK_STREAM, 0);
```

DESCRIPTION

IPv6 is the next generation network-layer protocol designed to be the successor to the current Internet Protocol version 4 (IPv4). It provides the packet delivery service for TCP, UDP and ICMPv6.

IPv6 has significant advantages over IPv4 in terms of increased address space, simplified header format, integrated QoS support and mandatory security. IPv6 also allows optional internet-layer information to be encoded in separate headers called extension headers which are placed between the IPv6 header and upper layer headers. Extension headers currently supported are hop-by-hop option header, destination option header, fragment header and routing (type 0) header. An IPv6 packet may carry zero, one, or more extension headers, each identified by the next header field of the preceding header.

IPv6 has extended the address size from 32 bits to 128 bits and they are textually represented in hex-colon notation as **x:x:x:x:x:x:x:x**, where the **x**'s are the hexadecimal values of the eight 16-bit pieces of the address. For example **fedc:83ff:fef6:417a:210:83ff:fef6:3dc0**.

IPv6 has three types of addresses: **unicast**, **anycast**, and **multicast**.

- An **unicast address** is an identifier for a single interface. A packet sent to an unicast address is delivered to the interface identified by that address.
- An **anycast address** is an identifier for a set of interfaces. A packet sent to an anycast address is delivered to one of the interfaces identified by that address.
- A **multicast address** is an identifier for a set of interfaces. A packet sent to a multicast address is delivered to all interfaces identified by that address.

  There are no broadcast addresses in IPv6, their function is superseded by multicast addresses.

Every IPv6 address has a **scope** associated with it. A scope is a topological span within which the address may be used as an unique identifier for an interface or set of interfaces.

An unicast address has three defined scopes: link-local, site-local and global.

- Link-local address uniquely identifies interfaces within a single link and it has a fixed prefix of **fe80::/10**. For example, **fe80::210:84c0:ef6f:cd30**.
- Site-local address uniquely identifies interfaces within a single site only and it has a fixed prefix of **fec0::/10**. For example, **fec0::210:84c0:ef6f:cd30**.
- Global address uniquely identifies interfaces anywhere in the internet.

There are 2 special unicast addresses which hold an embedded IPv4 address in the low order 32-bits.

- The first type is termed as IPv4-compatible IPv6 address and is of the form **0:0:0:0:0:0:d.d.d.d**. This type of address is used by dual stack (IPv4/IPv6) nodes to perform automatic IPv6-over-IPv4 tunneling where the IPv4 tunnel endpoint address is determined from the IPv4 address embedded in the IPv4-compatible destination address of the IPv6 packet being tunneled.
- The second type is termed as IPv4-mapped IPv6 address and is of the form **0:0:0:0:0:ffff:d.d.d.d**. This address facilitates IPv6 applications to interoperate with IPv4 applications. Applications can automatically generate this address using **getaddrinfo()** (see *getaddrinfo*(3N)) when the specified host has only IPv4 address.

IPv6 Socket Options

New socket options are defined for IPv6 to send and receive extension headers and to exchange other optional information between the kernel and application. The options are supported at the **IPPROTO_IPV6** protocol level. The type of the variable pointed to by the *optval* parameter is indicated in parenthesis.

IPV6_UNICAST_HOPS (**integer**) Set or get the hop limit used in outgoing unicast packets. When this option is set using **setsockopt()** (see *setsockopt*(2)), the new option value specified is used as the hop limit for all subsequent unicast packets sent via that socket. Valid values are in the range 0-255 (both inclusive) and the default value is 64. For example,

```
int hoplimit = 50;
setsockopt(s, IPPROTO_IPV6, IPV6_UNICAST_HOPS,
        &hoplimit, sizeof(hoplimit));
```

This option can be used with **getsockopt()** (see *getsockopt*(2)) to determine the hop limit value the system will use for subsequent unicast packets sent via that socket.

IPV6_MULTICAST_HOPS
(**integer**) Set or get the hop limit used in outgoing multicast packets. When this option is set, the new option value specified is used as the hop limit for all subsequent multicast packets sent via that socket. Valid values are in the range 0-255 (both inclusive) and the default value is 1.

IPV6_MULTICAST_IF (**integer**) Sets the interface to use for outgoing multicast packets. The option value is the index of the selected outgoing interface. For example,

```
unsigned int index;
index = if_nametoindex("lan0");
setsockopt(s, IPPROTO_IPV6, IPV6_MULTICAST_IF,
        &index, sizeof(index));
```

IPV6_MULTICAST_LOOP
(**boolean**) Enables or disables loopback in the IP layer for multicast datagrams sent through this socket. The value of the variable pointed to by *optval* is zero (disable) or non-zero (enable). Default: enabled.

IPV6_JOIN_GROUP (**struct ipv6_mreq**) Join a multicast group on a specified local interface. The IPv6 multicast address of the group to join and the index of the interface on which to join should be specified using **struct ipv6_mreq** which is defined in <**netinet/in6.h**> as:

```
struct ipv6_mreq {
   struct in6_addr ipv6mr_multiaddr;
                   /* IPv6 multicast addr */
   unsigned int  ipv6mr_interface;
                   /* interface index */
};
```

If the interface index is specified as 0 then the default multicast interface is used.

IPV6_LEAVE_GROUP (**struct ipv6_mreq**) Leave a multicast group on a specified local interface. The IPv6 multicast address of the group to leave and the interface index should be specified using **struct ipv6_mreq**. The interface index should match the index used while joining the group. Set index to 0, to specify default interface.

IPV6_CHECKSUM (**integer**) When this option is set, kernel computes the checksum for outbound packets and verifies checksum on inbound packets. The option value is the byte offset of the checksum location in the user data. This option is not valid for **IPPROTO_ICMPV6** since checksum computation is mandatory for **IPPROTO_ICMPV6**. The default value is -1 (checksums not computed nor verified for protocols other than **IPPROTO_ICMPV6**).

IPV6_RECVPKTINFO (**boolean**) When this option is enabled, **PKTINFO** (destination IPv6 address and the arriving interface index) is returned as ancillary data by **recvmsg()**. (See *recvmsg*(2)). The information is returned in **struct**

i

in6_pktinfo structure and it is defined in <**netinet/in6.h**>**as:**

```
struct in6_pktinfo {
      struct in6_addr ipi6_addr;
      uint32_t        ipi6_ifindex;
};
```

By default this option is disabled.

IPV6_RECVHOPLIMIT (**boolean**) When this option is enabled, inbound packet's hoplimit is returned as ancillary data by **recvmsg()**. For example,

```
int on = 1;
setsockopt(s, IPPROTO_IPV6, IPV6_RECVHOPLIMIT,
              &on, sizeof(on));
```

By default this option is disabled.

IPV6_RECVDSTOPTS (**boolean**) When this option is enabled, the inbound packet's destination options (when present) is returned as ancillary data by **recvmsg()**. By default this option is disabled.

IPV6_RECVHOPOPTS (**boolean**) When this option is enabled, the inbound packet's hop-by-hop options (when present) is returned as ancillary data by **recvmsg()**. By default this option is disabled.

i

IPV6_RECVRTHDR (**integer; boolean**) When this option is enabled, the inbound packet's routing options (when present) is returned as ancillary data by **recvmsg()**. By default this option is disabled.

IPV6_RECVRTHDRDSTOPTS
(**integer; boolean**) When this option is enabled, the inbound packet's destination options appearing before a routing header (when present) is returned as ancillary data by **recvmsg()**. By default this option is disabled.

The next seven socket options can be used with both **setsockopt()** and as option name in ancillary data to **sendmsg()**. (See *sendmsg*(2))

IPV6_PKTINFO (**struct in6_pktinfo**) Used to set the source address and interface index for outgoing packets.

IPV6_HOPLIMIT (**integer**) Used to set the hop limit for outbound packets. This hop limit is valid for only a single output operation. To set hop limit for all unicast or multicast IPv6 packets use **IPV6_UNICAST_HOPS** or **IPV6_MULTICAST_HOPS** options respectively.

IPV6_NEXTHOP (**struct sockaddr_in6**) Used to set the next hop address. The node identified by this address must be a neighbor of the sending host. When this address is the same as the destination IPv6 address then this is equivalent to **SO_DONTROUTE** socket option.

IPV6_RTHDR (**variable length**) Used to specify the routing header for outgoing packets. Only Type 0 routing header is currently supported.

IPV6_DSTOPTS (**variable length**) Used to specify one or more destination options to be sent in subsequent IPv6 packets.

IPV6_HOPOPTS (**variable length**) Used to specify one or more hop-by-hop options to be sent in subsequent IPv6 packets.

IPV6_RTHDRDSTOPTS (**variable length**) Used to specify one or more destination options preceding a routing header. This option will be silently ignored when sending packets unless a routing header is also specified.

IPv6 uses the enhanced version of ICMP called ICMPv6 to report errors encountered in processing packets and for diagnostic purposes (like ping). ICMPv6 is an integral part of IPv6 and has a next header value of 58.

All the options and the associated structures are defined in **<netinet/in6.h>**, applications are not required to include this header file explicitly, it is automatically included by **<netinet/in.h>**.

ERRORS

One of the following errors may be returned when a socket operation fails.

[EADDRINUSE] The specified multicast group has been joined already.

[EADDRNOTAVAIL] The specified IPv6 address is not a local interface address or there is no route for the specified multicast address or the specified multicast group has not been joined.

[EINVAL] The parameter 'level' is not **IPPROTO_IPV6**, or *optval* is the NULL address, or the specified multicast address is not valid, or the specified hop limit is not in the range 0 <= x<= 255.

[ENOBUFS] Insufficient memory is available for internal system data structures.

[ENOPROTOOPT] The parameter optname is not a valid socket option for the **IPPROTO_IPV6** level.

AUTHOR

The socket interfaces to IP were developed by the University of California, Berkeley.

SEE ALSO

bind(2), getsockopt(2), recv(2), send(2), socket(2), inet6_opt_init(3N), inet6_rth_space(3N), inet(7F), ndp(7P).

RFC 2460 *Internet Protocol Version 6*.
RFC 2553 *Basic Socket Interface Extensions for IPv6*.
RFC 2292 *Advanced Socket Interface Extensions for IPv6*.

i

NAME

kmem - perform I/O on kernel memory, based on symbol name

SYNOPSIS

```
#include <sys/ksym.h>

int ioctl(
    int kmemfd,
    int command,
    void *rks
    );
```

DESCRIPTION

When used with a valid file descriptor for **/dev/kmem** (*kmemfd*), **ioctl** can be used to manipulate kernel memory. The specifics of this manipulation depend on the *command* given as follows:

MIOC_READKSYM Read *mirk_buflen* bytes of kernel memory starting at the address for *mirk_symname* into *mirk_buf*. *rks* is a pointer to a **mioc_rksym** structure, defined below.

MIOC_IREADKSYM Indirect read. Read **sizeof(void *)** bytes of kernel memory starting at the address for *mirk_symname* and use that as the address from which to read *mirk_buflen* bytes of kernel memory into *mirk_buf*. *rks* is a pointer to a **mioc_rksym** structure.

MIOC_WRITEKSYM Write *mirk_buflen* bytes from *mirk_buf* into kernel memory starting at the address for *mirk_symname*. *rks* is a pointer to a **mioc_rksym** structure.

k

MIOC_IWRITEKSYM Indirect write. Read **sizeof(void *)** bytes of kernel memory starting at the address for *mirk_symname* and use that as the kernel memory address into which *mirk_buflen* bytes from *mirk_buf* are written. *rks* is a pointer to a **mioc_rksym** structure.

MIOC_LOCKSYM Increase the hold count by one for the dynamically loaded module whose name is given by *rks*, a pointer to a character string, thereby preventing its unloading.

MIOC_UNLOCKSYM Decrease the hold count by one for the dynamically loaded module whose name is given by *rks*, a pointer to a character string. If the count is thereby reduced to 0, the module becomes a candidate for unloading.

The **struct mioc_rksym** definition is:

```
struct mioc_rksym {
     char * mirk_modname;   /* limit search for symname
                               to module modname; if NULL
                               use standard search order */
     char * mirk_symname;   /* name of symbol whose address
                               is the basis for this
                               operation */
     void * mirk_buf;       /* buffer into/from which
                               read/write takes place */
     size_t mirk_buflen;    /* length (in bytes) of desired
                               operation */
};
```

RETURN VALUE

ioctl returns one of the following values:

0 Successful completion.

-1 Failure. **errno** is set to indicate the error.

ERRORS

In addition to the values described in *ioctl*(2), the **kmem ioctl** also sets **errno** to one of the following values if the corresponding condition is detected.

[EINVAL] *modname* does not represent a currently loaded module or this is an **MIOC_UNLOCKSYM** and the hold count is already 0.

[ENXIO] *kmemfd* open on wrong minor device (i.e., not **/dev/kmem**).

[EBADF] *kmemfd* open for reading and this is an **MIOC_WRITEKSYM**.

[ENOMATCH] *symname* not found.

[ENAMETOOLONG] *modname* is greater than **MODMAXNAMELEN** characters long, or *symname* is greater that **MAXSYMNMLEN** characters long.

SEE ALSO

getksym(2), ioctl(2), ioctl(5).

k

NAME

lan - network I/O card access information

DESCRIPTION

This manual entry gives a brief description on how to access the LAN device driver at Layer 2 (Data Link Layer) of the OSI architecture. The LAN device driver controls the various LAN interface cards (e.g, Ethernet/IEEE 802.3, FDDI, Token Ring) at Layer 1 (Physical Layer).

The Data Link Provider Interface (DLPI) is the supported method for accessing the LAN device driver at Layer 2. DLPI is intended for use by knowledgeable network users only. Refer to the *DLPI Programmer's Guide* for complete programming details.

There are HP and non-HP drivers and interface cards which will provide their own DLPI module. These types of DLPI are referred to as "native" DLPI.

Overview

The Physical Point of Attachment (PPA) is a numerical value that uniquely identifies a particular device. The PPA value can be obtained from the **nwmgr** and **lanscan** commands. The "ClassInstance" identifier in the **nwmgr** output is the concatenation of the driver class (lan) and the PPA number. The "NamePPA" identifier in the **lanscan** output is a concatenation of the interface name and the PPA number. The **card instance** value for a lan device is equivalent to the PPA number for that device.

A single hardware device may have multiple "NamePPA" identifiers, which indicates multiple encapsulation methods supported for to the device. For Ethernet/IEEE 802.3 links, the "Name" **lan** is used to designate Ethernet encapsulation, and **snap** for IEEE 802.3 encapsulation. For other links (FDDI, Token Ring), only the **lan** encapsulation designation is used.

Methods of transfer over the DLPI interface through the lan devices include "raw", "connectionless", and "connection-oriented" data transfers.

1

WARNINGS

The **lanadmin**, **lanscan** and **linkloop** commands are deprecated. These commands will be removed in a future HP-UX release. HP recommends the use of replacement command *nwmgr*(1M) to perform all network interface-related tasks.

AUTHOR

lan was developed by HP.

SEE ALSO

lanscan(1M), lanadmin(1M), linkloop(1M), nwmgr(1M).

DLPI Programmer's Guide, 1995, Hewlett-Packard

The Ethernet, A LAN: Data Link Layer and Physical Specification, Version 2.0, November 1982, Digital Equipment Corporation, Intel Corporation, Xerox Corporation

CSMA/CD Access Method and Physical Layer Specification, 1996, Institute of Electrical and Electronic Engineers

Demand-Priority Access Method, Physical Layer & Repeater Specifications, 1996, Institute of Electrical and Electronic Engineers

Fiber Distributed Data Interface (FDDI) Physical Layer Medium Dependent (PMD), 1995, ANSI

Token Ring Access Method and Physical Layer Specification, 1995, Institute of Electrical and Electronic Engineers

802.3u Media Access Control Parameters, Physical Layer, Medium Attachment Units, and Repeater for 100 Mb/s Operation, Type 100BASE-T, 1995, Institute of Electrical and Electronic Engineers

NAME

ldterm - standard STREAMS terminal line discipline module

SYNOPSIS

```
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/termios.h>
#include <sys/bsdtty.h>
#include <sys/ttold.h>
#include <sys/strtio.h>
#include <sys/eucioctl.h>

int ioctl( fd, I_PUSH, "ldterm");
```

DESCRIPTION

ldterm is a STREAMS module that supplies the line discipline for streams-based terminal or pseudo-terminal device drivers. This module provides most of the functions of the general terminal interface described in *termio*(7). However, it does not perform the low-level device control functions specified by the **c_cflag** word defined by the POSIX **termios** structure or the System V **termio** structure (defined in **termios.h** and **termio.h**, respectively). Also, some operations require the cooperation of the modules and drivers pushed below the **ldterm** module in a tty or pty (slave) stream. This man page only covers **ldterm** specific interface here and refers to the readers to *termio*(7) for the detail terminal interface.

Internally, the **ldterm** module uses the Extended UNIX Code (EUC) character encoding scheme. This encoding scheme enables the **ldterm** module to process multibyte characters as well as simple 8-bit characters. It correctly handles backspacing, word erasing, and tab expansion for multibyte EUC characters.

The **ldterm** module provides standard terminal operation consistent with the behavior specified by POSIX 1003.1 and System V Interface Definition (SVID) Third Edition. It also provides compatibility with the behavior of the BSD 4.3 line discipline. Notice that on other STREAMS systems, the BSD 4.3 compatibility feature is usually provided by a separate STREAMS module called **ttcompat**. Hence, applications on HP-UX need not push **ttcompat** on top of **ldterm** to get BSD 4.3 compatibility. In fact, the **ttcompat** module is not provided on the HP-UX system at all.

The **ldterm** module normally sits above either a STREAMS tty driver or a STREAMS pty slave driver. The user issues an **STREAMS I_PUSH** *ioctl*(2) system call to push **ldterm** onto the stream once the STREAMS tty or STREAMS pty slave device is opened.

STREAMS Messages

The **ldterm** module processes various types of STREAMS messages. The line discipline will act on any of the following message types. Any others that the module receives, however, are passed onto the next module on the stream.

Read-side Behavior

ldterm processes the following STREAMS messages on its input stream:

M_FLUSH
: If **FLUSHR** is set, the read put routine flushes the read queue, discards characters in the input message buffers, and discards any partially buffered multibyte EUC characters. Then, it forwards the message upstream.

M_BREAK
: The read put routine processes the message according to POSIX rules for processing **BREAK** events, parity errors, and framing errors and signal generation (see *termio*(7) for detail). If there is no data in the message, the message is assumed to represent an input **BREAK** event, which is represented by a framing error with a character value of 0 (zero). If there is data in the message, the data value is an integer that indicates the occurrence of an input **BREAK** event, or a character received with a parity or framing error. The low-order 8 bits of the data value is the byte that was read. If the **TTY_PE** flag is set in the higher-order bits of this integer, then a parity error was detected. If the **TTY_FE** flag is set in the higher-order bits of this integer, a framing error was detected.

  After reading the data value, the read put routine discards the message.

M_DATA
: The read put routine processes the message according to the POSIX 1003.1 specification, using multibyte processing for backspacing, word erasing, and tab expansion as appropriate.

It generates echo characters and places them in the output buffer to be sent downstream to the write queue. While processing incoming data, it scans for **START** and **STOP** characters and sends **M_START**, **M_STOP** messages downstream to the write queue, if needed.

If the total number of buffered input characters is more than the high-water mark and **IXOFF** is set, the read put routine sends an **M_STOPI** message downstream. When the queue reduces its backlog below the low water mark, it sends an **M_STARTI** message downstream.

If the number of buffered input characters reaches **MAX_INPUT**, and the **IMAXBEL** flag is set, the read put routine discards new input characters and sends a **BEL** character (**Ctrl-G**) downstream. If **IMAXBEL** is not set, it flushes the input queue.

If the **ISIG** flag is set, the read put routine sends **M_PCSIG** messages upstream when the appropriate signal characters are encountered. Then it discards the characters.

If a character matching **c_cc[VDISCARD]** is encountered, and the **IEXTEN** flag is set, the read put routine sends an **M_FLUSH** (**FLUSHW**) message upstream to flush all write queues. The **M_FLUSH** message is reflected by the stream head and sent downstream through all the write queues.

If the character signifies the logical termination of input, the read put routine sends the currently buffered characters upstream to the stream head.

Logical termination of input depends on the state of the **ICANON** flag. If **ICANON** is set, the **ldterm** module is in canonical input mode. In that case, the read put routine logically terminates input at the end of a line of input. Canonical line termination characters are **NEWLINE**, **EOF**, **EOL**, and **EOL2**. If **ICANON** is clear, the **ldterm** discipline module is in noncanonical or raw input mode. In that case, the read put routine terminates input when at least **VMIN** bytes are present in the input message buffer or the timer specified by **VTIME** expires (see *termio*(7) for more details).

1

M_IOCACK

If the message acknowledges the POSIX **termios TCGETS** command, the read put routine copies the **c_cflag** and speeds information, which is sent by the console driver downstream, from the message into the internal POSIX **termios** structure. Then it copies the internal POSIX **termios** structure into the message.

If the message acknowledges one of the POSIX **termios** set commands (i.e. **TCSETS**, **TCSETSW**, and **TCSETSF**) the read put routine copies all of the data from the message into the internal POSIX **termios** structure.

After this processing is done, the read put routine determines if the I/O control command was originally a BSD 4.3 or System V I/O control command that was converted to a POSIX **termios** command by the write service routine. If so, it restores the original data so that the message acknowledges the original I/O control command. Then it forwards the message upstream.

M_CTL This message was sent by the driver to make special requests to **ldterm.** The structure of **M_CTL** messages is the same as that of **M_IOCTL** messages. The **M_CTL** message block points to a message buffer containing an **iocblk** data structure (defined in **<sys/stream.h>**). The **ioc_cmd** member of this structure contains a command, just as it does in an **M_IOCTL** message. The **b_cont** member of the **M_CTL** message block contains a pointer to an **M_DATA** message block, which contains data associated with the **M_CTL** message.

The read put routine processes **M_CTL** messages containing the following commands:

MC_NO_CANON

Turn off input processing normally performed on upstream **M_DATA** messages. This is for the use of modules or drivers that perform their own input processing such as pseudo-terminal (see *ptm*(7) and *pts*(7)) in **REMOTE** mode connected to a program that performs the input processing.

MC_DO_CANON

Turn on input processing normally performed on upstream **M_DATA** messages. This message is sent when the driver want **ldterm** to exit the **REMOTE** mode.

Write-side Behavior

ldterm processes the following STREAMS messages on its output stream. Messages not listed here are simply forwarded downstream.

M_FLUSH
The write put routine flushes the write queue and discards any buffered output data. Then, it forwards the message downstream.

M_DATA The write service routine processes the data according to the POSIX 1003.1 specification output flags. It sends the processed characters downstream to the driver when the output queue fills up and all of the data is processed.

M_IOCTL
The write put routine validates the format of the **M_IOCTL** message and checks for known commands. If the message format is invalid, it turns the **M_IOCTL** message into an **M_IOCNAK** message, and returns the message upstream. If the I/O control command is not recognized, it forwards the **M_IOCTL** message downstream for processing by other modules.

The write put routine determines if the command is one that must be processed in the proper sequence relative to **M_DATA** messages. If so, it queues the **M_IOCTL** message to the write queue for later processing by the write service routine. Commands that require processing in sequence are:

TCSETSW, **TCSETSF**, **TCSETAW**, **TCSETAF**, **TCSBRK**

Otherwise, the module's write put routine processes the command immediately. Detailed descriptions of the preceding **ioctl** commands are provided in the *ioctl* Commands subsection, below.

M_READ This message is sent by the stream head to notify downstream modules when an application has issued a read request and there is not enough data queued at the stream head to satisfy the request. The **M_READ** is sent downstream normally when **ldterm** is operating in non-canonical input mode. If **VTIME** is positive, the write put routine starts an input timer. When the timer expires, it sends all buffered input upstream. Then, it forwards the **M_READ** message downstream.

1

ioctl Commands

The **ldterm** module acts on two categories of **ioctl** commands:

- Primary terminal I/O control commands
- BSD 4.3 compatibility terminal I/O control commands

Detail descriptions on how to use these **ioctls** can be found on the *termio*(7) man page. **NOTE:** the **FIO[xyz] ioctls** documented on *termio*(7) are currently not supported on **ldterm**.

Primary Terminal I/O Control Commands

The **ldterm** module acts on the following primary terminal I/O commands:

TCSETS, **TCSETSW**, **TCSETSF**
When the **ldterm** module receives any of these commands in an **M_IOCTL** message, it forwards them downstream. When it receives the **M_IOCACK** message in the read queue, it copies the POSIX **termios** information from the message into the internal POSIX **termios** structure and forwards the message upstream. If a mode change requires options at the stream head to be changed, an **M_SETOPTS** message is sent upstream. If the **ICANON** flag is turned on or off, the read mode at the stream head is changed to message-nondiscard (**RMSGN**) with read notification on (**SO_MREADON**) or byte-stream mode (**RNORM**) with read notification off (**SO_MREADOFF**), respectively. If the **TOSTOP** flag is turned on or off, the tostop mode at the stream head is turned on (**SO_TOSTOP**) or off (**SO_TONSTOP**), respectively.

TCGETS The **ldterm** module forwards the **M_IOCTL** message downstream. When it receives the **M_IOCACK** message in the read queue, it copies the **CLOCAL** flags and speeds from the message into the internal POSIX **termios** structure. Then, it copies the entire structure into the **M_IOCACK** message and forwards the message upstream.

TCSETA, **TCSETAW**, **TCSETAF**
These commands set the old System V **termio** information. The **ldterm** module converts the message to a POSIX **termios M_IOCTL** message, then forwards the message with a corresponding POSIX **termios** command (i.e. **TCSETS**, **TCSETSW**, **TCSETSF**). The original I/O control command and **M_IOCTL** message are stored for use on **M_IOCACK**.

TCGETA This command get the old System V **termio** information. The **ldterm** module converts the message to a POSIX **termios M_IOCTL** message, then forwards the message with the

TCGETS command. The original I/O control command and **M_IOCTL** message are stored to be used on **M_IOCACK**. When it receives the matching **M_IOCACK** message, the **ldterm** module processes it as for a **TCGETS** command, then converts the POSIX **termios** information into the System V **termio** information and replies.

TCSBRK The **ldterm** module forwards this command downstream to be handled by the driver so that the driver has a chance to drain the data before sending an **M_IOCACK** message upstream.

TCXONC This command controls the behavior of input/output flow control. If the argument is 0 and output is not already stopped, an **M_STOP** message is sent downstream. If the argument is 1 and the output is stopped, an **M_START** message is sent downstream. If the argument is 2 and input is not already stopped, an **M_STOPI** message is sent downstream. If the argument is 3 and input is stopped, an **M_STARTI** message is sent downstream.

TCFLSH This command flush the input or/and output streams. If the argument is 0, an **M_FLUSH** message with a flag byte of **FLUSHR** is sent downstream. This **M_FLUSH** (**FLUSHR**) message will be reflected back upstream by the driver to flush the entire input stream. If the argument is 1, an **M_FLUSH** message with a flag byte of **FLUSHW** is sent upstream. This **M_FLUSH** (**FLUSHW**) message will be reflected downstream by the stream head to flush the entire output stream.

TIOCSWINSZ
This command sets the window size variables. The argument of this command takes a pointer to a **winsize** structure. The **ldterm** module does not use the window size variable, but maintains it here for any needed replies to **TIOCGWINSZ** commands. The module forwards the message downstream.

TIOCGWINSZ
When the **ldterm** module receives this command, it returns the window size variable that was set by the last **TIOCSWINSZ** command. The argument of this command takes a pointer to a **winsize** structure.

1

EUC_WSET
This command sets the character widths and screen widths for the EUC character sets. The argument of this command takes a pointer to an **eucioc** structure which contains the information for setting the character widths and screen widths of the EUC character sets. After processing the command, **ldterm** forwards this message downstream to the next module.

EUC_WGET
This command returns the character widths and screen widths for the EUC character sets. This command takes a pointer to an **eucioc** structure via which the EUC character widths and screen widths information will be returned.

EUC_SET_HP15
This command put **ldterm** to the so called **HP15** mode which enable **ldterm** to recognize the HP15_SJIS, HP15_BIG5, HP15_CCDC, and HP15_GB character sets and process them in such a way that they behave like EUC characters. The argument for this command takes a pointer to an integer value which specify on of the above-mentioned four supported HP15 character sets. If the argument is set to HP15_ASCII, then **ldterm** will switch back to normal ASCII processing. **EUC_WSET** is mutually exclusive with **EUC_SET_HP15**.

EUC_GET_HP15
This command returns the current HP15 character that has been set via the **EUC_SET_HP15** command. This command takes a pointer to an integer via which the result is returned. If no previous **EUC_SET_HP15** has been issued, then it will return HP15_ASCII.

BSD 4.3 Compatible Terminal I/O Commands

The **ldterm** module acts on the following I/O commands, which are compatible with the BSD I/O environment:

TIOCEXCL
Set 'exclusive-use' mode. No further opens are permitted until the file has been closed.

TIOCNXCL
Turn off 'exclusive-use' mode.

TIOCSETD
The **ldterm** module does nothing but reply to this command. In a BSD system, the command is used to set the current line discipline type. It does not have much meaning in a STREAMS

environment, because line discipline modules are changed by popping the current module from the stream and pushing a different one onto the stream.

TIOCGETD
In a BSD system, this command is used to get the current line discipline type. The command does not have much meaning in a STREAMS environment. The **ldterm** module replies with a value of 2 for binary compatibility, since **ldterm** supports job control.

TIOCFLUSH
This command flush the input or/and output streams similar to that of the **TCFLSH** command. The argument is a pointer to an **int** variable. If its value is zero, both the input and output streams are flushed by sending the appropriate **FLUSHR**/**FLUSHW** **M_FLUSH** messages upstream and downstream. Otherwise, the value of the **int** is treated as the logical **OR** of the **FREAD** and **FWRITE** flags defined by **<sys/file.h>**. If the **FREAD** flag is set, the input stream is flushed. If the **FWRITE** flag is set, the output stream is flushed. Then, **ldterm** acknowledges the message with **M_IOCACK**.

TIOCOUTQ
This command takes a pointer to an integer and returns the number of characters buffered up in the **ldterm's** output buffer.

TIOCHPCL
This command sets the POSIX **termios HUPCL** flag to indicate that the terminal line should be disconnected when the last file descriptor associated with that line is closed. The **ldterm** module converts the command into a compatible POSIX **termios** I/O control command by sending an **M_IOCTL** message containing the **TCSETS** command with current **termios** settings downstream.

TIOCSTART
The command restarts output. If the terminal was stopped, the **ldterm** module sends an **M_START** message downstream.

TIOCSTOP
This command stops output. The **ldterm** module sends an **M_STOP** message downstream.

TIOCSBRK
This command sets the break condition on a line. The **ldterm** module sends an **M_BREAK** message containing a value of 1 as data to the driver, then replies with **M_IOCACK**

TIOCCBRK
This command clears the break condition on a line. The **ldterm** module sends an **M_BREAK** message containing a value of 0 (zero) as data to the driver, then replies with **M_IOCACK**.

TIOCSETP, **TIOCSETN**
These commands set the **sgttyb** information, defined in **<sys/ttold.h>**. The argument is a pointer to an **sgttyb** structure. The **ldterm** module converts the message to a POSIX **termios M_IOCTL** message. Then, it forwards the POSIX **termios M_IOCTL** message with a corresponding POSIX **termios** command (i.e. **TCSETSW**, **TCSETS**). The original I/O control command and **M_IOCTL** message are stored for use on **M_IOCACK**.

TIOCGETP
This command returns the **sgttyb** information based on the interpretation of the current content of the POSIX **termios** structure maintained in **ldterm**. The argument is a pointer to an **sgttyb** structure via where the information is returned.

TIOCSETC
This command sets the **tchars** information, defined in **<sys/strtio.h>**. The argument is a pointer to an **tchars** structure. The **ldterm** module converts the message to a POSIX **termios M_IOCTL** message. Then, it forwards the POSIX **termios M_IOCTL** message with a corresponding POSIX **termios** command (i.e. **TCSETS**). The original I/O control command and **M_IOCTL** message are stored for use on **M_IOCACK**.

TIOCGETC
This command returns the **tchars** information based on the interpretation of the current content of the POSIX **termios** structure maintained in **ldterm**. The argument is a pointer to an **tchars** structure via where the information is returned.

TIOCSLTC
This command sets the **ltchars** information defined in **<sys/bsdtty.h>**. The **ldterm**

module converts the message to a POSIX **termios M_IOCTL** message. Then, it forwards the POSIX **termios M_IOCTL** message with a corresponding POSIX **termios** command (i.e. **TCSETS**). The original I/O control command and **M_IOCTL** message are stored for use on **M_IOCACK**.

TIOCGLTC
The **ldterm** module returns the **ltchars** information based on the interpretation of the current content of the POSIX **termios** structure maintained in **ldterm**.

TIOCLBIS, **TIOCLBIC**, **TIOCLSET**
These commands set the BSD 4.3 flags information, defined in **<sys/strtio.h>**. For **TIOCLBIS** and **TIOCLBIC**, the argument is a pointer to an **int** whose value is a mask containing flags to be set/clear. For **TIOCLSET**, the argument is a pointer to an **int** whose value is a new set of flags to be set. The **ldterm** module converts the message to a POSIX **termios M_IOCTL**, then forwards the POSIX **termios M_IOCTL** message with a corresponding POSIX **termios** command (i.e. **TCSETS**). It stores the original I/O control command and **M_IOCTL** message to be used on **M_IOCACK**.

TIOCLGET
The **ldterm** module returns the BSD 4.3 flags information based on the interpretation of the current content of the POSIX **termios** structure maintained in **ldterm**.

TIOCSTI
This command takes an argument of a pointer to a character and pretends that the character was typed on the terminal. The user must either have the **DEVOPS** privilege or have read permission on the controlling terminal against which the ioctl is issued. See *privileges*(5) for more information about privileged access on systems that support fine-grained privileges.

FIONREAD
This command takes an argument of a pointer to an integer and returns the number of immediately readable characters.

AUTHOR

ldterm was developed by HP and OSF.

SEE ALSO

ioctl(2), privileges(5), ptem(7), ptm(7), pts(7), streamio(7), termio(7).

NAME

lp - line printer

SYNOPSIS

#include <sys/lprio.h>

Remarks

This manual entry applies only to a certain group of printers. For Series 800, it applies to printers controlled by the device driver **lpr2**. It does *not* apply to any printers on Series 700 systems.

DESCRIPTION

This section describes capabilities provided by many line printers supported by various versions of the HP-UX operating system. A line printer is a character special device that may optionally have an interpretation applied to the data.

If the character special device file has been created with the raw option (see the HP-UX System Administrator manuals for information about creating device files with the raw option), data is sent to the printer in **raw mode** (as, for example, when handling a graphics printing operation). In raw mode, no interpretation is done on the data to be printed, and no page formatting is performed. Data bytes are simply sent to the printer and printed exactly as received.

If the device file does not contain the raw option, data can still be sent to the printer in raw mode. Raw mode is set and cleared by the **LPRSET** request.

If the line printer device file does not contain the raw option, data is interpreted according to rules discussed below. The driver understands the concept of a printer page in that it has a page length (in lines), line length (in characters), and offset from the left margin (in characters). The default line length, indent, lines per page, open and close page eject, and handling of backspace are set to defaults determined when the printer is opened and recognized by the system the first time. If the printer is not recognized, the default line length is 132 characters, indent is 4 characters, lines per page is 66, one page is ejected on close and none on open, and backspace is handled for a character printer.

The following rules describe the interpretation of the data stream:

- A form feed causes a page eject and resets the line counter to zero.
- Multiple consecutive form-feeds are treated as a single form-feed.
- The new-line character is mapped into a carriage-return/line-feed sequence, and if an offset is specified a number of blanks are inserted after the carriage-return/line-feed sequence.
- A new-line that extends over the end of a page is turned into a form-feed.
- Tab characters are expanded into the appropriate number of blanks (tab stops are assumed to occur every eight character positions as offset by the current indent value).
- Backspaces are interpreted to yield the appropriate overstrike either for a character printer or a line printer.
- Lines longer than the line length minus the indent (i.e., 128 characters, using the above defaults) are truncated.
- Carriage-return characters cause the line to be overstruck.
- When it is opened or closed, a suitable number of page ejects is generated.

Two *ioctl*(2) requests are available to control the lines per page, characters per line, indent, handling of backspaces, and number of pages to be ejected at open and close times. At either open or close time, if no page eject is requested the paper will not be moved. For opens, line and page counting will start assuming a top-of-form condition.

The *ioctl* requests have the following form:

```
#include <sys/lprio.h>

int ioctl(int fildes, int request, struct lprio *arg);
```

The possible values of *request* are:

LPRGET Get the current printer status information and store in the **lprio** structure to which *arg* points.

1

LPRSET Set the current printer status information from the structure to which *arg* points.

The **lprio** structure used in the **LPRGET** and **LPRSET** requests is defined in **<sys/lprio.h>**, and includes the following members:

```
short int  ind;        /* indent */
short int  col;        /* columns per page */
short int  line;       /* lines per page */
short int  bksp;       /* backspace handling flag */
short int  open_ej;    /* pages to eject on open */
short int  close_ej;   /* pages to eject on close */
short int  raw_mode;   /* raw mode flag */
```

These are remembered across opens, so the indent, page width, and page length can be set with an external program. If the **col** field is set to zero, the defaults are restored at the next open.

If the backspace handling flag is 0, a character printer is assumed and backspaces are passed through the driver unchanged. If the flag is a 1, a line printer is assumed, and sufficient print operations are generated to generate the appropriate overstruck characters.

If the raw mode flag is 0, data sent to the printer is formatted according to indent, columns per page, lines per page, backspace handling, and pages to eject on open and close.

If the raw mode flag is 1, data sent to the printer is not formatted.

If the raw mode flag is changed from 1 to 0 (raw mode is turned off) and the format settings (indent, columns per page, etc.) have not been modified, the data is formatted according to the prior format settings.

AUTHOR

lp was developed by HP and AT&T.

1

FILES

/dev/lp default or standard printer used by some HP-UX commands;
/dev/[r]lp* special files for printers

SEE ALSO

lp(1), slp(1), ioctl(2), cent(7), intro(7).

NAME

lvm - Logical Volume Manager (LVM)

DESCRIPTION

The Logical Volume Manager (LVM) is a subsystem for managing disk space. The HP LVM subsystem offers value-added features, such as mirroring (with the optional HP MirrorDisk/UX software), high availability (with the optional HP ServiceGuard software), and striping, that enhance availability and performance.

Unlike earlier arrangements where disks were divided into fixed-sized sections, LVM allows the user to consider the disks, also known as **physical volumes**, as a pool (or volume) of data storage, consisting of equal-sized extents. The default size of an extent is 4 MB.

An LVM system consists of arbitrary groupings of physical volumes, organized into **volume groups**. A volume group can consist of one or more physical volumes. There can be more than one volume group in the system. Once created, the volume group, and not the disk, is the basic unit of data storage. Thus, whereas earlier one would move disks from one system to another, with LVM, one would move a volume group from one system to another. For this reason it is often convenient to have multiple volume groups on a system.

Volume groups can be subdivided into virtual disks, called **logical volumes**. A logical volume can span a number of physical volumes or represent only a portion of one physical volume. The pool of disk space that is represented by a volume group can be apportioned into logical volumes of various sizes. The size of a logical volume is determined by its number of extents. Once created, logical volumes can be treated just like disk partitions. Logical volumes can be assigned to file systems, used as swap or dump devices, or used for raw access.

Commands

LVM information can be created, displayed, and manipulated with the following commands:

lvchange	Change logical volume characteristics
lvcreate	Stripe, create logical volume in volume group
lvdisplay	Display information about logical volumes
lvextend	Increase space, increase mirrors for logical volume
lvlnboot	Prepare logical volume to be root, primary swap, or dump volume
lvreduce	Decrease number of physical extents allocated to logical volume
lvremove	Remove one or more logical volumes from volume group
lvrmboot	Remove logical volume link to root, primary swap, or dump volume
pvchange	Change characteristics of physical volume in volume group
pvcreate	Create physical volume for use in volume group
pvdisplay	Display information about physical volumes within volume group
pvmove	Move allocated physical extents from one physical volume to other physical volumes
vgcfgbackup	Create or update volume group configuration backup file
vgcfgrestore	Display or restore volume group configuration from backup file
vgchange	Set volume group availability
vgcreate	Create volume group
vgdisplay	Display information about volume groups
vgexport	Export a volume group and its associated logical volumes
vgextend	Extend a volume group by adding physical volumes
vgimport	Import a volume group onto the system
vgmodify	Modify volume group attributes
vgreduce	Remove physical volumes from a volume group
vgremove	Remove volume group definition from the system
vgscan	Scan physical volumes for volume groups

The following commands are also available if the HP MirrorDisk/UX software is installed:

lvmerge	Merge two logical volumes into one logical volume
lvsplit	Split mirrored logical volume into two logical volumes
lvsync	Synchronize stale mirrors in logical volumes
vgsync	Synchronize stale logical volume mirrors in volume groups

Device Special Files

In this release of HP-UX 11i, the Mass Storage Stack supports two naming conventions for the device special files used to identify devices (see *intro*(7)). Devices can be represented using:

- Persistent device special files, (**/dev/disk/disk3**), or
- Legacy device special file names, (**/dev/dsk/c0t6d6**).

While LVM supports the use of both conventions within the same volume group, the examples shown in the LVM man pages are all using the legacy device special file convention.

Alternate Links (PVLinks)

In this release of HP-UX, LVM continues to support Alternate Links to a device to allow continued access to the device, if the primary link fails. This multiple link or multipath solution increases data availability, but continues disallowing the use of multiple paths simultaneously.

A new feature was introduced in the Mass Storage Subsystem on HP-UX 11i Version 3 that supports multiple paths to a device and allows simultaneous access to these paths. The Mass Storage Subsystem will balance the I/O load across the valid paths. Multipathing is the default unless the **scsimgr** command is used to enable legacy multipathing and also the active path is a legacy device special file. See *scsimgr*(1M) for details.

Even though the Mass Storage Subsystem supports 32 multiple paths per physical volume on this version of HP-UX, LVM does not support more than eight paths to any physical volume. As a result, commands like **vgcreate** and **vgextend** will not succeed in adding more than eight paths per physical volume. Additionally, **vgimport** and **vgscan** cannot write more than eight paths per physical volume in the **/etc/lvmtab** file. If users want to use any specific path other than these eight paths, then they have to **vgreduce** one of the alternate paths in the volume group and add that specific path using **vgextend**.

It is no longer required or recommended to configure LVM with alternate links. However, it is possible to maintain the traditional LVM behavior. To do so, both of the following criteria must be met:

- Only the legacy device special file naming convention is used in the volume group configuration.
- The **scsimgr** command is used to enable the legacy multipath behavior for each physical volume in the volume group.

EXAMPLES

The basic steps to take to begin using LVM are as follows:

- Identify the disks to be used for LVM.
- Create an LVM data structure on each identified disk (see *pvcreate*(1M)).
- Collect all the physical volumes to form a new volume group (see *vgcreate*(1M)).
- Create logical volumes from the space in the volume group (see *lvcreate*(1M)).
- Use each logical volume as if it were a disk section (create a file system, or use for raw access).

To configure disk **/dev/dsk/c0t0d0** as part of a new volume group named **vg01**:

First, initialize the disk for LVM with the **pvcreate** command.

```
pvcreate /dev/rdsk/c0t0d0
```

Then, create the pseudo device file that is used by the LVM subsystem.

```
mkdir /dev/vg01
mknod /dev/vg01/group c 64 0x010000
```

The minor number for the **group** file should be unique among all the volume groups on the system. It has the format **0x***NN***0000**, where *NN* ranges from **00** to **ff**.

Create the volume group, **vg01**, containing the physical volume, **/dev/dsk/c0t0d0**, with the **vgcreate** command.

```
vgcreate /dev/vg01 /dev/dsk/c0t0d0
```

You can view information about the newly created volume group with the **vgdisplay** command.

```
vgdisplay -v /dev/vg01
```

Create a logical volume of size 100 MB, named **usrvol**, on this volume group with the **lvcreate** command.

```
lvcreate -L 100 -n usrvol /dev/vg01
```

This creates two device files for the logical volume, **/dev/vg01/usrvol**, which is the block device file, and **/dev/vg01/rusrvol**, which is the character (raw) device file.

You can view information about the newly created logical volume with the **lvdisplay** command.

```
lvdisplay /dev/vg01/lvol1
```

Any operation allowed on a disk partition is allowed on the logical volume. Thus, you can use **usrvol** to hold a file system.

```
newfs /dev/vg01/rusrvol
mount /dev/vg01/usrvol /usr
```

SEE ALSO

lvchange(1M), lvcreate(1M), lvdisplay(1M), lvextend(1M), lvlnboot(1M), lvreduce(1M), lvremove(1M), lvrmboot(1M), pvchange(1M), pvcreate(1M), pvdisplay(1M), pvmove(1M), vgcfgbackup(1M), vgcfgrestore(1M), vgchange(1M), vgcreate(1M), vgdisplay(1M), vgexport(1M), vgextend(1M), vgimport(1M), vgmodify(1M), vgreduce(1M), vgremove(1M), vgscan(1M), intro(7).

Managing Systems and Workgroups.

If HP MirrorDisk/UX is installed: lvmerge(1M), lvsplit(1M), lvsync(1M), vgsync(1M).

If HP ServiceGuard is installed: cmcheckconf(1M), cmquerycl(1M), *Managing MC/ServiceGuard*.

1

NAME

mem - main memory image file

DESCRIPTION

mem is a special file that is an image of the main memory of the computer. It may be used, for example, to examine and patch the system.

Byte addresses in **mem** are interpreted as physical memory addresses. References to non-existent locations cause errors to be returned.

File **kmem** is the same as **mem** except that kernel virtual memory rather than physical memory is accessed. Please refer to *kmem*(7) for information about ioctl operations that are supported on **/dev/kmem**.

WARNINGS

Examining and patching device registers is likely to lead to unexpected results when read-only or write-only bits are present.

FILES

/dev/mem

/dev/kmem

SEE ALSO

kmem(7).

NAME

modem - asynchronous serial modem line control

SYNOPSIS

#include <sys/modem.h>

DESCRIPTION

This section describes the two modes of modem line control and the three types of terminal port access. It also discusses the effect of the bits of the *termio* structure that affect modem line control. The modem-related **ioctl()** system calls (see *ioctl*(2)) are discussed at the end of the manpage.

Definitions

There are several terms that are used within the following discussion which will be defined here for reference.

"Modem control lines" (CONTROL) are generally defined as those outgoing modem lines that are automatically controlled by the driver.

"Modem status lines" (STATUS) are generally defined as those incoming modem lines that are automatically monitored by the driver.

CONTROL and STATUS for a terminal file vary according to the modem line control mode of the file (see the *Modem Line Control Modes* section below).

An **open()** (see *open*(2)) to a port is considered to be BLOCKED if it is waiting for another file on the same port to be closed.

An **open()** to a port is considered to be PENDING if it is waiting for the STATUS to be raised.

An **open()** to a port is considered to be SUCCESSFUL if the **open()** system call has returned to the calling process without error.

Open Flag Bits

Currently, the only **open()** flag bits recognized by the driver are the **O_NDELAY** and **O_NONBLOCK** bits. When either of these bits is set, an **open()** call to the driver will never become blocked. If possible, the **open()** will be returned immediately as SUCCESSFUL, and the driver will continue the process of opening the tty file. If it is not possible, then the **open()** will be returned immediately with the appropriate error code as described in the appropriate section.

m

Termio Bits

When set, the **CLOCAL** bit in the *termios* or *termio* structure (see *termio*(7)) is used to remove the driver's automatic monitoring of the modem lines. However, the user's ability to control the modem lines is determined only by the mode in effect and does not depend on the state of **CLOCAL**. Normally, the driver will monitor and require the STATUS to be raised. An **open()** system call will raise the CONTROL and wait for the STATUS before completing unless the **CLOCAL** bit is set. (If the **O_NDELAY** or **O_NONBLOCK** bit is set, the **open()** will be returned immediately, but the driver will otherwise continue to monitor the modem lines as normal based on the state of the **CLOCAL** bit.) Normally, loss of the STATUS will cause the driver to break the modem connection and lower the CONTROL. However, if **CLOCAL** is set, any changes in the STATUS will be ignored. A connection is required before any data may be read or written, unless **CLOCAL** is set. Any timers that would normally be in effect (see the *Modem Line Control Modes* and *Modem Timers* sections below) will be stopped while **CLOCAL** is set.

When the **CLOCAL** bit is changed from clear to set, the driver will assume the existence of an active device (such as a modem) on the port regardless of the STATUS. If any of the CONTROL are raised at that point in time, they will continue in that state. The STATUS will no longer be actively monitored. When the **CLOCAL** bit is changed from set to clear, the driver will resume actively monitoring the STATUS. If all of the CONTROL and STATUS are raised at that point in time, the driver will continue the modem connection. If any of the STATUS are not raised, the driver will act as though those signals were lost (as described in the *Modem Line Control Modes* section below) and, if the device is a controlling terminal, a *hangup* signal will be sent to the controlling process. If any of the CONTROL are not raised, the driver will break the modem connection by lowering all the CONTROL.

The **HUPCL** bit in the *termios* or *termio* structure determines the action of the driver regarding the CONTROL when the last **close()** system call (see *close*(2)) is issued to a terminal file. If the HUPCL bit is set, the driver will lower the CONTROL at **close()** time and the modem connection will be broken. If **HUPCL** is not set and a modem connection exists, it will continue to exist, even after the **close()** is issued. The driver will not change the CONTROL.

Terminal Port Access Types

There are three types of modem access: call-in connections, call-out connections, and direct (no modem control) connections. A given port may be accessed through all three types of connection by accessing different files. The modem access type of a terminal file is determined by the file's major and/or minor device numbers.

The call-in type of access is used when the connection is expected to be established by an incoming call. This is the type that would be used by *getty*(1M) to accept logins over a modem. When an **open()** is issued to such a file, the driver may wait for an incoming call and will then raise the CONTROL based on the current mode (see below) of the port. When the port is closed, the driver may or may not lower the CONTROL depending on the **HUPCL** bit.

The call-out type of access is used when the connection is expected to be established by an outgoing call. This would be used by programs such as *uucp*(1). When an **open()** is issued to such a file, the driver will immediately raise the CONTROL and wait for a connection based on the mode currently in effect. When the port is closed, the driver may or may not lower the CONTROL depending on the **HUPCL** bit.

The direct type of access is used when no driver modem control is desired. This could then be used for directly connected terminals that use a three-wire connection, or to talk to a modem before a connection has been established. The second case allows a program to give dialing instructions to the modem. Neither the **CLOCAL** nor the **HUPCL** bits have any effect on a port accessed through a direct file. (However, both bits may be inherited by other types of files; see the *Terminal Port Access Interlock* section below.) An **open()** to a direct file does not affect the CONTROL and does not depend on any particular state of the STATUS to succeed. When the file is closed, the driver will not affect the state of the CONTROL. If a modem connection has been established, it will continue to exist. Setting the speed of a direct file to B0 (see *termio*(7)) will be considered an impossible speed change and will be ignored. It will not affect the CONTROL.

Modem Line Control Modes

There are two modes of modem line control: CCITT mode and simple mode. A given port may have only one of these two modes in effect at any given point in time. An attempt to open a port with a mode other than the one in effect (from a PENDING or SUCCESSFUL **open()** on a different file) will cause the **open()** to be returned with an [ENXIO] error. The modem access type of a terminal file is determined by the file's major and/or minor device numbers.

CCITT mode is used for connections to switched line modems. The CONTROL for CCITT mode are Data Terminal Ready (DTR) and Request to Send (RTS). The STATUS are Data Set Ready (DSR), Data Carrier Detect (DCD), and Clear to Send (CTS). Additionally, the Ring Indicator (RI) signal indicates the presence of an incoming call. When a connection is begun (an incoming call for a call-in file or an **open()** issued to a call-out file), the CONTROL are raised and a connection timer (see the *Modem Timers* section below) is started. If the STATUS become raised before the time period has elapsed, a connection is established and the **open()** request is returned successfully. If the time period expires, the CONTROL are lowered and the connection is aborted. For a call-in file, the driver will wait for another incoming call; for a call-out file, the **open()** will be returned with an [EIO] error. Once a connection is established, loss of either DSR or CTS will cause the CONTROL to be lowered and, if the device is a controlling terminal, a *hangup* signal will be sent to the controlling process.

If DCD is lost, a timer is started. If DCD resumes before the time period has expired, the connection will be maintained. However, no data transfer will occur during this time. The driver will stop transmitting characters, and any characters received by the driver will be discarded. (However, on some implementations data transmission cannot be stopped. See the *DEPENDENCIES* section.) If DCD is not restored within the allotted time, the connection will be broken as described above for DSR and CTS.

If the modem connection is to be broken when the **close()** system call is issued (i.e. **HUPCL** is set), then the CONTROL will be lowered and the **close()** will be returned as successful. However, no further **open()**s will be allowed until after both DSR and CTS have been lowered by the modem, and the hangup timer (see the *Modem Timers* section below) has expired. The action taken in response to an **open()** during this time will be the same as if the port were still open. (See the *Terminal Port Access Interlock* section below.)

When a port is in CCITT mode, the driver has complete control of the modem lines and the user is not allowed to change the setting of the CONTROL or affect which STATUS are actively monitored by the driver (see the *Modem Ioctls* section below). This is to provide strict adherence with the CCITT recommendations.

Simple mode is used for connections to devices which require only a simple method of modem line control. This can include devices such as black boxes, data switches, or for system-to-system connections. It can also be used with modems which cannot operate under the CCITT recommendations. The CONTROL for simple mode consists of only DTR. The STATUS consists of only DCD. When an **open()** is issued, the CONTROL is raised but no connection timer is started. When the STATUS becomes raised, a connection is established and the **open()** request is returned as SUCCESSFUL. Once a connection is established, loss of the STATUS will cause the CONTROL to be lowered and, if the device is a controlling terminal, a *hangup* signal will be sent to the controlling process.

When a port is in simple mode, the driver will normally control the modem lines. However, the user is allowed to change the setting of the CONTROL (see the *Modem Ioctls* section below).

Terminal Port Access Interlock

An interlock mechanism is provided between the three access types of terminal files. It prevents more than one file from being successfully opened at a time, but allows certain **open()**s to succeed while others are PENDING so that a port can be opened through a call-out connection while *getty* has a pending **open()** at a call-in connection. The three access types are given a priority that determines which **open()** will succeed if more than one file has an **open()** issued against it. The three access types are ordered from lowest priority to highest as follows: call-in, call-out, and direct.

If an **open()** is issued to a port which already has a SUCCESSFUL **open()** on it of a lower priority type, the new **open()** will be returned with an [EBUSY] error. ([EBUSY] will also be returned by an attempted **open()** on a CCITT call-out file if an incoming call indication is currently being received. In this case, if there is a PENDING **open()** on the corresponding CCITT call-in file, this PENDING **open()** will complete.) If the lower priority **open()** is PENDING, the new **open()** will succeed if possible, or will be left PENDING if waiting for the STATUS and the lower priority **open()** will become BLOCKED. If a higher priority **open()** has succeeded or is PENDING, the new **open()** will be BLOCKED, unless the new **open()** has the **O_NDELAY** flag bit set, in which case the **open()** will be returned with an [EBUSY] error. Once an **open()** on one type of file is SUCCESSFUL, any PENDING *open*s on lower priority files will become BLOCKED.

When a file of one priority is closed, a BLOCKED **open()** on the next lower priority type file will become active. If all of the STATUS are raised, the **open()** will be SUCCESSFUL, otherwise the **open()** will become PENDING waiting for the STATUS. If the lower priority **open()** is SUCCESSFUL (because the connection was maintained when the higher priority file was closed), the port characteristics (speed, parity, etc.) that were set by the higher priority file will be inherited by the lower priority file. If the connection is not maintained through the **close()**, the port characteristics will be set to default values.

Modem Timers

There are four timers currently defined for use with modem connections. The first three of the timers are applicable only to CCITT mode connections. In general, the effect of changing a timer value while the timer is running is system dependent. However, setting the timer value to zero is guaranteed to disable the timer even if it is running.

The connect timer is used to limit the amount of time to wait for a connection to be established once it has been begun. This timer is started when an incoming call has been received on a call-in file, or when an **open()** has been issued on a call-out file for which no *open*s are already pending. If the connection is completed in time, the timer is aborted. If the time period expires, the connection is aborted. For a call-in file, the driver will again wait for an incoming call and the **open()** will remain pending. For a call-out file, the **open()** will be returned with an [EIO] error.

The carrier detect timer is used to limit the amount of time to wait before causing a disconnect if DCD drops. If carrier is not re-established in this time, a disconnect will occur. If carrier is re-established before the timeout, the timer will be aborted and the connection maintained. During the period when carrier is not raised, no data will be transferred across the line.

The no activity timer is used to limit the amount of time a connection will remain open with no data transfer across the line. When the data line becomes quiescent with no data transfer, this timer will be started. If data is again transferred over the line in either direction before the time limit, the timer will be aborted. If no activity occurs before the timeout has occurred, the driver will disconnect the line. This can be used to avoid long and costly telephone connections when data transfer has been stopped either normally or abnormally.

The last timer defined, the hangup timer, is used for both CCITT and simple modes. This timer controls the amount of time to wait after disconnecting a modem line before allowing another **open()**. This time period should be made long enough to guarantee that the connection has been terminated by the telephone switching equipment. If this period is not long enough, the telephone connection may not be broken and a succeeding **open()** may complete with the old connection.

HP-UX Modem Ioctls

Several **ioctl()** system calls apply to manipulation of modem lines. They use the following information defined in **<sys/modem.h>**:

```
#define  NMTIMER   6
typedef  unsigned  long   mflag;
struct   mtimer {
         unsigned  short  m_timers[NMTIMER];
         };
```

Each bit of the *mflag* long corresponds to one of the modem lines as follows:

MRTS	Request to Send	outbound
MCTS	Clear to Send	inbound
MDSR	Data Set Ready	inbound
MDCD	Data Carrier Detect	inbound
MDTR	Data Terminal Ready	outbound
MRI	Ring Indicator	inbound
MDRS	Data Rate Select	outbound

The timer values are defined in the array **m_timers**. The relative position of the timer and default initial values and units for each timer are as follows:

0	**MTCONNECT**	25 s
1	**MTCARRIER**	400 ms
2	**MTNOACTIVITY**	0 min
3	**MTHANGUP**	250 ms
4	**Reserved**	
5	**Reserved**	

A value of zero for any timer will disable that timer.

The modem line **ioctl()** system calls have the form:

int ioctl(int *fildes***, int** *command***, mflag ****arg***);**

The commands using this form are:

MCGETA Get the current state of both inbound and outbound modem lines and store in the **mflag** long referenced by **arg**. A raised line will be indicated by a one bit in the appropriate position.

MCSETA Set the outbound modem lines from the *mflag* long referenced by **arg**. Setting an outbound bit to one causes that line to be raised and zero to be lowered. Setting bits for inbound lines has no effect. Setting any bits while in CCITT mode has no effect. The change to the modem lines is immediate and using this form while characters are still being output may cause unpredictable results.

MCSETAW Wait for the output to drain and set the new parameters as described above.

MCSETAF Wait for the output to drain, then flush the input queue and set the new parameters as described above.

The timer value **ioctl()** system calls have the form:

int ioctl(int *fildes***, int** *command***, mtimer ****arg***);**

The commands using this form are:

MCGETT Get the current timer value settings and store in the *mtimer* structure referenced by **arg**.

MCSETT Set the timer values from the structure referenced by **arg**.

For any timer, setting the timer value to its previous value has no effect.

SVID3 Modem Ioctls

System V Interface Definition, Third Edition (SVID3) specifies additional **ioctl()** system calls to manipulate the modem lines. They use information defined in **<termios.h>**.

Each **ioctl()** passes an integer argument in which each of the following bit definitions correspond to one of the modem lines as follows:

TIOCM_RTS	Request to Send	outbound
TIOCM_CTS	Clear to Send	inbound
TIOCM_DSR	Data Set Ready	inbound
TIOCM_CAR	Data Carrier Detect	inbound
TIOCM_DTR	Data Terminal Ready	outbound
TIOCM_RNG	Ring Indicator	inbound

Additionally, **TIOCM_CD** is equivalent to **TIOCM_CAR**, and **TIOCM_RI** is equivalent to **TIOCM_RNG**.

The modem line **ioctl()** system calls have the form:

int ioctl(int *fildes***, int** *command***, int ***arg***);**

The commands using this form are:

TIOCMGET Get the current state of both inbound and outbound modem lines and store in the int referenced by **arg**. A raised line will be indicated by a one bit in the appropriate position.

TIOCMSET Set the outbound modem lines from the int referenced by **arg**.

TIOCMBIS Raise the control lines specified by a one in the corresponding bit positions of the int referenced by **arg**.

TIOCMBIC Lower the control lines specified by a one in the corresponding bit positions of the int referenced by **arg**.

Note that setting bits for inbound lines has no effect, and setting any bits while in CCITT mode has no effect. Also, the change to the modem lines is immediate and using these ioctl's while characters are still being output may cause unpredictable results.

WARNINGS

Occasionally it is possible that a process may open a call-out file at approximately the same time as an incoming call is received. In some cases, the call-out connection may be satisfied by the incoming call. In general, however, the results are indeterminate. If necessary, the situation can be avoided by the use of two modems and ports, one for call-out connections and the other for receiving incoming calls.

DEPENDENCIES

Some hardware implementations may not have access to all modem lines supported by MCSETA. If a particular hardware does not support a given line, attempts to set the value of a line will be ignored, and reading the current state of the line will return zero. The appropriate I/O card manual should be referenced to determine the lines supported by the hardware installed.

Some hardware implementations may not have access to all timers supported by MCSETT. Also, the granularity of the individual timers may vary depending on the hardware and system in use. The effect of setting a timer out of range or with a granularity outside the capability of a particular system should be documented by that system. The effect of changing the value for a timer while that timer is running is system dependent and should be documented by each system.

Setting the **CLOCAL** bit while a timer is running will cause the timer to be stopped. It is a system dependency whether or not the timer is restarted, and if so, the value at which it is restarted when the **CLOCAL** bit is subsequently cleared.

On those implementations supporting the HP27140A 6-Channel Multiplexer, transmission of characters cannot be stopped during loss of DCD. The driver cannot detect loss of DCD until the connection is broken. Also, the I/O card may still have characters in its internal buffers and will still try to transmit them.

AUTHOR

modem was developed by HP and AT&T.

FILES

/dev/cua*
/dev/cul*
/dev/tty*
/dev/ttyd*

SEE ALSO

stty(1), mknod(1M), ioctl(2), open(2), termio(7).

NAME

mt - magnetic tape interface and controls for stape and estape

DESCRIPTION

This entry describes the behavior of HP magnetic tape interfaces and controls. The files **/dev/rtape/*** refer to specific raw tape drives controlled by the estape driver. The major number of these device special files is dynamically allocated and the minor number does not encode any device specific information.

The files **/dev/rmt/*** refer to specific raw tape drives controlled by the legacy stape driver, and the behavior of each given unit is specified in the major and minor numbers of the DSF. The legacy driver and DSFs are deprecated and will be removed in a future version of HP-UX.

Naming Conventions

The device special files (referred to as DSFs) for the **estape** driver have the following naming conventions:

/dev/rtape/tape#_BEST[**n**][**b**]

There are four such files (referred to as persistent DSFs) corresponding to each of the four different permutations of the **n** and **b** options. These are claimed by the **estape** driver. See *intro*(7) for more details on persistent device special file names.

There are two naming conventions for legacy DSFs. The standard (preferred) convention is used on systems that support long file names. An alternate convention is provided for systems limited to short file names. The following standard convention is recommended because it allows for all possible configuration options in the device name and is used by *mksf*(1M) and *insf*(1M):

/dev/rmt/c#**t**#**d**#[**o**][**z**][**e**][**p**][**s**[#]][**w**]**BEST**[**C**[#]][**n**][**b**]

The following alternate naming convention is provided to support systems in which the **/dev/rmt** directory requires short file names. These DSF names are less descriptive, but guarantee unique device naming and are used by *mksf*(1M) and *insf*(1M) where required.

/dev/rmt/c#**t**#**d**#[**f**#|**i**#][**n**][**b**]

For each tape device present, twelve DSFs are automatically created when the system is installed. If legacy mode is disabled (via the **-L** option in **rmsf**), only four DSFs in **/dev/rtape** will be created post installation. These are claimed by the **estape** driver.

Four legacy DSFs will be created in the **/dev/rmt** directory using the following naming convention. These are legacy DSFs and are claimed by the **stape** driver.

/dev/rmt/c#**t**#**d**#**BEST**[**n**][**b**].

Four more legacy DSFs with the format **/dev/rmt/**#**m**[**n**][**b**] will be automatically created when the system is installed using the pre-HP-UX 10.0 device file naming convention. This includes an arbitrary number to distinguish this tape device from others in the system, followed by the letter **m**. There are four such DSFs because each of the four different permutations of the **n** and **b** options (see below) are created. These files are created for compatability with pre-HP-UX 10.0 scripts and for users who find the old convention easier to remember.

Each of the automatically created DSFs which utilize the standard or alternate naming conventions is linked to a device file which utilizes the pre-HP-UX 10.0 naming convention. That is, the DSFs in the format **/dev/rmt/**#**m**[**n**][**b**] are created as hardlinks to the corresponding **/dev/rmt/c**#**t**#**d**#**BEST**[**n**][**b**] DSFs mentioned above.

Thus, the DSFs which utilize the pre-HP-UX 10.0 naming convention provide the same functionality as the device files which contain the density specification **BEST** (standard naming convention).

Options

The options described here are common to all legacy tape drivers. The **c**#**t**#**d**# notation in the legacy DSF name derives from **ioscan** output and is described in the manpages for *ioscan*(1M) and *intro*(7).

c# Instance number assigned by the operating system to the interface card.

t# Target address on a remote bus (for example, SCSI address)

d# Device unit number at the target address (for example, SCSI LUN).

w Writes wait for physical completion of the operation before returning status. The default behavior (buffered mode or immediate reporting mode) requires the tape device to buffer the data and return immediately with successful status.

density Density or format used in writing data to tape. This field is designated by the following values:

BEST Highest-capacity density or format will be used, including data compression, if the device supports compression.

NOMOD Maintains the density used for data previously written to the tape. Behavior using this option is dependent on the type of device. This option is only supported on DDS drives.

DDS Selects one of the known DDS formats; can be used to specify **DDS1** or **DDS2**, as required.

DLT Selects one of the known DLT formats; can be used to specify **DLT42500_24**, **DLT42500_56**, **DLT62500_64**, **DLT81633_64**, or **DLT85937_52**, as required.

D[#] Specifies density as a numeric value to be placed in the SCSI mode select block descriptor. The header file **<sys/mtio.h>** contains a list of the standard density codes. The numeric value is used only for density codes which *cannot* be found in this list.

C[#] Write data in compressed mode, on tape drives that support data compression. If a number is included, use it to specify a compression algorithm specific to the device. Note, compression is also provided when the density field is set to **BEST**.

n No rewind on close. Unless this mode is requested, the tape is automatically rewound upon close.

b Specifies Berkeley-style tape behavior. When the **b** is absent, the tape drive follows AT&T-style behavior. The details are described in *Tape Behavioral Characteristics* below.

f# Specify format (or density) value encoded in the minor number. The meaning of the value is dependent on the type of tape device in use. (Used for short file name notation only.)

m

i# Specify an internal Property Table index value maintained by the tape driver, containing an array of configuration options. The contents of this table are not directly accessible. Use the *lssf*(1M) command to determine which configuration options are invoked. (Used for short file name notation only.)

o Console message disabled. See *mksf*(1M).

z RTE compatible close. See *mksf*(1M).

e Exhaustive mode. See *DEPENDENCIES* section.

p Tape partition. See *DEPENDENCIES* section.

s Fixed-block mode. See *DEPENDENCIES* section.

#**m** For pre-HP-UX 10.x device file naming convention.

Sample Tape Device Special File Names

For a HP Ultrium-2 drive at card instance 1, target 2, LUN 3 the legacy DSFs would be **/dev/rmt/c1t2d3BEST**[**n**][**b**]. The corresponding persistent DSFs assuming an instance number "1" allocated to the DSF would be **/dev/rtape/tape1_BEST**[**n**][**b**]. Corresponding device special files in the pre-HP-UX 10.0 naming convention would be **/dev/rmt/0m**[**n**][**b**]. In this particular example, 0 (zero) in **0m**[**n**][**b**] denotes an instance number of 0 (zero) assigned to the DSF. The files in the **/dev/rmt/**#**m**[**n**][**b**] format are created as hardlinks to the corresponding **/dev/rmt/c**#**t**#**d**#**BEST**[**n**][**b**] DSFs.

Use the *lssf*(1M) command to determine which configuration options are actually used with any device file. The naming convention defined above should indicate the options used, but device files may be created with any user defined name.

Tape Behavioral Characteristics

When opened for reading or writing, the tape is assumed to be positioned as desired.

When a file opened for writing is closed, two consecutive EOF (End of File) marks are written if, and only if, one or more writes to the file have occurred. The tape is rewound unless the no-rewind mode has been specified, in which case the tape is positioned before the second EOF just written.

When a file open for reading (only) is closed and the no-rewind bit is not set, the tape is rewound. If the no-rewind bit is set, the behaviour depends on the *style* mode. For AT&T-style devices, the tape is positioned after the EOF following the data just read (unless already at BOT or Filemark). For Berkeley-style devices, the tape is not repositioned in any way.

Each *read*(2) or *write*(2) call reads or writes the next record on the tape. For writes, the record has the same length as the buffer given (within the limits of the hardware).

During a read, the record size is passed back as the number of bytes read, up to the buffer size specified. Since the minimum read length on a tape device is a complete record (to the next record mark), the number of bytes ignored (for records longer than the buffer size specified) is available in the **mt_resid** field of the **mtget** structure via the **MTIOCGET** call of *ioctl*(2). Current restrictions require tape device application programs to use 2-byte alignment for buffer locations and I/O sizes. To allow for more stringent future restrictions (4-byte aligned, etc.) and to maximize performance, page alignment is suggested. For example, if the target buffer is contained within a structure, care must be taken that structure elements before the buffer allow the target buffer to begin on an even address. If need be, placing a filler integer before the target buffer will insure its location on a 4-byte boundary.

The ascending hierarchy of tape marks is defined as follows: record mark, filemark (EOF), setmark and EOD (End of Data). Not all devices support all types of tape marks but the positioning within the hierarchy holds true. Each type of mark is typically used to contain one or more of the lesser marks.

When spacing over a number of a particular type of tape mark, hierarchically superior marks (except EOD) do not terminate tape motion and are included in the count. For instance, MTFSR can be used to pass over record marks and filemarks.

Reading an EOF mark is returned as a successful zero-length read; that is, the data count returned is zero and the tape is positioned after the EOF, enabling the next read to return the next record.

DDS devices also support setmarks, which are used to delineate a group (set) of files. Reading a setmark is also returned as a zero-length read. Filemarks, setmarks and EOD can be distinguished by unique bits in the **mt_gstat** field.

Spacing operations (back or forward space, setmark, file or record) position past the object being spaced to in the direction of motion. For example, back-spacing a file leaves the tape positioned before the file mark; forward-spacing a file leaves the tape positioned after the file mark. This is consistent with standard tape usage.

lseek(2) type seeks on a magnetic tape device are ignored. Instead, the *ioctl*(2) operations below can be used to position the tape and determine its status.

The header file **<sys/mtio.h>** has useful information for tape handling.

Macros to Decode Options

The minor number of the device ID (**dev_t**) of persistent tape device special files no longer encode the tape device options (such as, density, style of access and so on). Hence the macros given below, that are defined in **<sys/mtio.h>** header file do not interpret the options correctly for persistent (agile) DSFs. The macros are:

```
M_INSTANCE(dev)          M_TARGET(dev)
M_LUN(dev)               M_BERKELEY(dev)
M_NO_REWIND(dev)         M_USER_CONFIG(dev)
M_INDEX(dev)             M_INDEX_PUT(dev,index)
M_DFLT_DENSITY(dev)      M_DFLT_DENSITY_PUT(dev,density)
M_TRANSPARENT_MODE(dev)  M_PROP_TBL_ACCESS(dev)
```

These macros continue to work on the legacy DSFs as before.

Applications should use the method described below to decode the tape device options from persistent device files.

libIO(3X) API **io_dev_to_options** is used to decode the device options from the persistent device files as given below:

```
#include <libIO.h>
#include <sys/_inttypes.h>
#include <fcntl.h>
```

Note: **libIO** calls should be within calls to **io_init()** and **io_end()**. Refer to *libIO*(3X) manpage for more details. Applications have to link with libIO library to access these APIs.

m

mt_get_newdev_options() and **mt_check_newdev_options()** are utility functions used by the code snippets below.

```
uint64_t
mt_get_newdev_options(dev_t dev, int dev_type) {
      uint64_t        options;
      int             err;
      err = io_dev_to_options(dev, dev_type, &options);
      if (err == IO_ERROR)
              return 0;
      return (options);
}

uint64_t
mt_check_newdev_options(dev_t dev, int dev_type, uint64_t bitmask) {
      uint64_t        options;
      int             err;
      err = io_dev_to_options(dev, dev_type, &options);
      if (err == IO_ERROR)
              return 0;
      return (options & bitmask);
}
```

For example, the macro **M_BERKELEY_AGILE** given below decodes the device options of both legacy and persistent (agile) DSFs. This macro returns true if the device ID is that of a device special file supporting Berkeley style of access.

Example

```
File test.c :

#include <stdlib.h>
#include <sys/libIO.h>
#include <sys/_inttypes.h>
#include <sys/stat.h>
#include <sys/errno.h>
#include <fcntl.h>
#include <sys/mtio.h>

#define MT_IS_LEGACY_DEV 1

#define M_BERKELEY_AGILE(dev)                                       \
          ((io_is_legacy_dev(dev, D_CHR) == MT_IS_LEGACY_DEV) ?    \
          (dev & MT_BSD_MASK)                              :       \
          (mt_check_newdev_options(dev, D_CHR, MT_BSD_MASK)))

/ *
  * It is assumed that definitions of mt_get_newdev_options() and
  * mt_check_newdev_options() are defined by the application and
  * available. Omitted here for the sake of simplicity.
  */

      int
      main(int argc, char *argv[]) {

            struct stat stbuf;
            dev_t dev;

            /* Device special file is passed as argv[1] */

            if (stat(argv[1], &stbuf) < 0)
            {
                perror("stat(): ");
                exit (1);
```

m

```
        }

        dev = stbuf.st_rdev;

        io_init(O_RDWR);

        if(M_BERKELEY_AGILE(dev))
            printf(" This is a Berkeley style device file ");
        else
            printf(" This is not a Berkeley style device file ");

        io_end();

        exit(0);
    }

Compile Line: cc -Ae -o test test.c -lIO

Sample Output:
# ./test /dev/rtape/tape1_BESTn
This is not a Berkeley style device file
# ./test /dev/rtape/tape1_BESTb
This is a Berkeley style device file
# ./test /dev/rmt/0mnb
This is a Berkeley style device file
# ./test /dev/rmt/c5t4d0BEST
This is not a Berkeley style device file
# ./test /dev/rmt/c5t4d0BESTnb
This is a Berkeley style device file
```

Macros similar to the one above, can be written in place of their respective legacy macros as follows:

```
#define M_INSTANCE_AGILE(dev)                                           \
        ((io_is_legacy_dev(dev, D_CHR) == MT_IS_LEGACY_DEV)    ?        \
        (((dev) >> MT_INSTANCE_BIT_POS) & MT_INSTANCE_MASK) :           \
        (((mt_get_newdev_options(dev, D_CHR)) >> MT_INSTANCE_BIT_POS) \
                                                  & MT_INSTANCE_MASK))

#define M_TARGET_AGILE(dev)                                             \
        ((io_is_legacy_dev(dev, D_CHR) == MT_IS_LEGACY_DEV)    ?        \
        (((dev) >> MT_TARGET_BIT_POS) & MT_TARGET_MASK)     :           \
        ((mt_get_newdev_options(dev, D_CHR)) >> MT_TARGET_BIT_POS)      \
                                                 & MT_TARGET_MASK))

#define M_LUN_AGILE(dev)                                                \
        ((io_is_legacy_dev(dev, D_CHR) == MT_IS_LEGACY_DEV)   ?         \
        (((dev) >> MT_LUN_BIT_POS) & MT_LUN_MASK)          :            \
        ((mt_get_newdev_options(dev, D_CHR) >> MT_LUN_BIT_POS)          \
                                              & MT_LUN_MASK))

#define M_USER_CONFIG_AGILE(dev)                                         \
        ((io_is_legacy_dev(dev, D_CHR) == MT_IS_LEGACY_DEV) ?            \
        (dev & MT_USER_CONFIG_MASK)                       :              \
        (mt_check_newdev_options(dev, D_CHR, MT_USER_CONFIG_MASK)))

#define M_INDEX_AGILE(dev)                                               \
        ((io_is_legacy_dev(dev, D_CHR) == MT_IS_LEGACY_DEV) ?            \
        (((dev) & MT_INDEX_MASK) >>  MT_INDEX_BIT_POS)    :              \
        ((mt_check_newdev_options(dev, D_CHR, MT_INDEX_MASK)) >>         \
                                               MT_INDEX_BIT_POS));

#define M_INDEX_PUT_AGILE(dev,index)                                     \
        ((io_is_legacy_dev(dev, D_CHR) == MT_IS_LEGACY_DEV) ?            \
```

m

```
        (((dev) & (~MT_INDEX_MASK))  |                                \
        (index << MT_INDEX_BIT_POS) |                                 \
         MT_USER_CONFIG_MASK)                           :             \
        ((mt_check_newdev_options(dev, D_CHR, ~MT_INDEX_MASK)) |      \
         (index << MT_INDEX_BIT_POS)))

#define M_DFLT_DENSITY_PUT_AGILE(dev,density)                         \
        ((io_is_legacy_dev(dev, D_CHR) == MT_IS_LEGACY_DEV) ?         \
        (((dev) & (~MT_DENSITY_MASK))    |                            \
        (density << MT_DENSITY_BIT_POS))                :             \
        ((mt_check_newdev_options(dev, D_CHR, ~MT_DENSITY_MASK))  |   \
                                   (density << MT_DENSITY_BIT_POS)))

#define M_TRANSPARENT_MODE_AGILE(dev)                                 \
        ((io_is_legacy_dev(dev, D_CHR) == MT_IS_LEGACY_DEV) ?         \
        (((dev) & MT_TRANSPARENT_MASK) ==                             \
                                MT_TRANSPARENT_VAL)      :            \
        ((mt_check_newdev_options(dev, D_CHR, MT_TRANSPARENT_MASK))   \
                                  == MT_TRANSPARENT_VAL))
```

The following is included from **<sys/mtio.h>** and describes the possible tape operations:

```
/* mag tape I/O control requests */

#define  MTIOCTOP  _IOW('m', 1, struct mtop)  /* do mag tape op */
#define  MTIOCGET  _IOR('m', 2, struct mtget) /* get tape status */

/* structure for MTIOCTOP - mag tape op command */

struct mtop {
    short mt_op;         /* operations defined below */
    int32_t mt_count;    /* how many of them */
};

/* operations */

#define MTWEOF 0  /* write filemark (end-of-file record) */
#define MTFSF  1  /* forward space file */
#define MTBSF  2  /* backward space file */
#define MTFSR  3  /* forward space record */
#define MTBSR  4  /* backward space record */
#define MTREW  5  /* rewind */
#define MTOFFL 6  /* rewind and put the drive offline (may eject) */
#define MTNOP  7  /* no operation, may set status */
#define MTEOD  8  /* DDS, QIC and 8MM only - seek to end-of-data */
#define MTWSS  9  /* DDS and 8MM only - write setmark(s) */
#define MTFSS 10  /* DDS and 8MM only - space forward setmark(s) */
#define MTBSS 11  /* DDS and 8MM only - space backward setmark(s) */
#define MTSTARTVOL 12  /* Start a new volume (for ATS) */
#define MTENDVOL 13  /* Terminate a volume (for ATS) */
#define MTRES 14  /* Reserve Device */
#define MTREL 15  /* Release Device */
#define MTERASE 16  /* Erase media */

/* structure for MTIOCGET - mag tape get status command */

struct mtget {
     long       mt_type;     /* type of magtape device */
     long       mt_resid;    /* residual count */

/* The following two registers are device dependent */
```

m

```
        long       mt_dsreg1;    /* status register (msb) */
        long       mt_dsreg2;    /* status register (lsb) */

    /* The following are device-independent status words */

        long       mt_gstat;     /* generic status */
        long       mt_erreg;     /* error register */
        int32_t    mt_fileno;    /* No longer used - always set to -1 */
        int32_t    mt_blkno;     /* No longer used - always set to -1 */
```

Information for decoding the **mt_type** field can be found in **<sys/mtio.h>**.

Tape operations work the same way for both legacy and agile devices.

Other Tape Status Characteristics

Efficient use of streaming tape drives with large internal buffers and immediate-reporting require the following end-of-tape procedures:

All writes near LEOT (Logical End of Tape) complete without error if actually written to the tape. Once the tape driver determines that LEOT has been passed, subsequent writes do not occur and an error message is returned.

To write beyond this point (keep in mind that streaming drives have already written well past LEOT), simply ask for status using the **MTIOCGET** ioctl. If status reflects the EOT condition, the driver drops all write barriers.

Both the **estape** and **stape** drivers will flush the device buffers when a write filemark (all devices) or write setmark (devices that support setmarks) command is given with the count set to zero.

When immediate-reporting is disabled, the write encountering LEOT returns an error with the tape driver automatically backing up over that record.

When reading near the end-of-tape, the user is not informed of LEOT. Instead, the typical double EOF marks or a pre-arranged data pattern signals the logical end-of-tape.

Since magnetic tape drives vary in EOT sensing due to differences in the physical placement of sensors, any application (such as multiple-tape *cpio*(1) backups) requiring that data be continued from the EOT area of one tape to another tape must be restricted. Therefore, the tape drive type and mode should be identical for the creation and reading of the tapes.

The following macros are defined in **<sys/mtio.h>** for decoding the status field mt_gstat returned from **MTIOCGET**. For each macro, the input parameter *x* is the **mt_gstat** field.

GMT_BOT(*x***)** — Returns TRUE at beginning of tape.

GMT_EOD(*x***)** — Returns TRUE if End-of-Data is encountered for DDS, QIC or 8MM.

GMT_EOF(*x***)** — Returns TRUE at an End-of-File mark.

GMT_EOT(*x***)** — Returns TRUE at end of tape.

GMT_IM_REP_EN(*x***)** — Returns TRUE if immediate reporting mode is enabled.

GMT_ONLINE(*x***)** — Returns TRUE if drive is online.

GMT_SM(*x***)** — Returns TRUE if setmark is encountered.

GMT_WR_PROT(*x***)** — Returns TRUE if tape is write protected.

GMT_COMPRESS(*x***)** — Returns TRUE if data compression is enabled.

GMT_DENSITY(*x***)** — Returns the currently configured 8-bit density value. Supported values are defined in **<sys/mtio.h>**.

GMT_D_800(*x***)** — Returns TRUE if the density encoded in **mt_gstat** is 800 bpi.

GMT_D_1600(*x***)** — Returns TRUE if the density encoded in **mt_gstat** is 1600 bpi.

GMT_D_6250(*x***)** — Returns TRUE if the density encoded in **mt_gstat** is 6250 bpi (with or without compression).

GMT_D_6250c(*x***)** — Returns TRUE if the density encoded in **mt_gstat** is 6250 bpi plus compression.

m

GMT_D_DDS1(*x*)	Returns TRUE if the density encoded in **mt_gstat** is DDS1 (with or without compression).
GMT_D_DDS1c(*x*)	Returns TRUE if the density encoded in **mt_gstat** is DDS1 plus compression.
GMT_D_DDS2(*x*)	Returns TRUE if the density encoded in **mt_gstat** is DDS2 (with or without compression).
GMT_D_DDS2c(*x*)	Returns TRUE if the density encoded in **mt_gstat** is DDS2 plus compression.
GMT_D_DLT_42500_24(*x*)	Returns TRUE if the density encoded in **mt_gstat** is 42500 bpi, 24 track pairs.
GMT_D_DLT_42500_56(*x*)	Returns TRUE if the density encoded in **mt_gstat** is 42500 bpi, 56 track pairs.
GMT_D_DLT_62500_64(*x*)	Returns TRUE if the density encoded in **mt_gstat** is 62500 bpi (with or without compression).
GMT_D_DLT_62500_64c(*x*)	Returns TRUE if the density encoded in **mt_gstat** is 62500 bpi plus compression.
GMT_D_DLT_81633_64(*x*)	Returns TRUE if the density encoded in **mt_gstat** is 81633 bpi (with or without compression).
GMT_D_DLT_81633_64c(*x*)	Returns TRUE if the density encoded in **mt_gstat** is 81633 bpi plus compression.
GMT_D_DLT_85937_52(*x*)	Returns TRUE if the density encoded in **mt_gstat** is 85937 bpi (with or without compression).
GMT_D_DLT_85937_52c(*x*)	Returns TRUE if the density encoded in **mt_gstat** is 85937 bpi plus compression.
GMT_D_3480(*x*)	Returns TRUE if the density encoded in **mt_gstat** is for a 3480 device (with or without compression).
GMT_D_3480c(*x*)	Returns TRUE if the density encoded in **mt_gstat** is for a 3480 device with compression.
GMT_DR_OPEN(*x*)	Does not apply to any currently supported devices. Always returns FALSE.

HP-UX silently enforces a tape record blocking factor (**MAXPHYS**) on large I/O requests. For example, a user write request with a length of ten times **MAXPHYS** will actually reach the media as ten separate records. A subsequent read (with ten times **MAXPHYS** as a length) will look like a single operation to the user, even though HP-UX has broken it up into ten separate read requests to the driver. The blocking function is transparent to the user during writes. It is also transparent during reads unless:

- The user picks an arbitrary read length greater than **MAXPHYS**.
- The user attempts to read a third-party tape containing records larger than **MAXPHYS**.

Since the value for **MAXPHYS** is relatively large (usually >= 256K bytes), this is typically not a problem.

The **MTNOP** operation does not set the device-independent status word.

EXAMPLES

Assuming that *fd* is a valid file descriptor, the following example writes two consecutive filemarks on the tape:

```
#include <sys/types.h>
#include <sys/mtio.h>

struct  mtop mtop;

mtop.mt_op = MTWEOF;
mtop.mt_count = 2;
ioctl(fd, MTIOCTOP, &mtop);
```

If *fd* is a valid file descriptor for an open DDS drive, the following example spaces forward to just past the next setmark:

```
#include <sys/types.h>
#include <sys/mtio.h>

struct  mtop mtop;

mtop.mt_op = MTFSS;
mtop.mt_count = 1;
ioctl(fd, MTIOCTOP, &mtop);
```

Given that *fd* is a valid file descriptor for an opened tape device, and that it has just returned 0 from a *read*(2) request. The following system call verifies that the tape has just read a filemark:

```
#include <sys/types.h>
#include <sys/mtio.h>

struct mtget mtget;

ioctl(fd, MTIOCGET, &mtget);
if (GMT_EOF (mtget.mt_gstat)) {
/* code for filemark detection */
}
```

WARNINGS

Density specification **BEST** (standard naming convention) activate data compression on tape devices which support compression. This is also true for the files using the pre-HP-UX 10.0 naming convention which are linked to these files (see "Naming Conventions" above).

For the persistent tape DSFs the minor number does not encode any configuration option. The minor number represents an index into a persistent kernel database where the configuration options are stored.

It is recommended that all legacy tape device files be put in the **/dev/rmt** directory. Legacy Device files using extended configuration options located outside the **/dev/rmt** directory may not provide consistent behavior across system reboots.

Although persistent DSFs may be created in directories other than **/dev/rtape**, HP recommends that persistent tape DSFs only be created in **/dev/rtape**.

Use the *rmsf*(1M) command to clean up unused device files. Otherwise, the property table may overflow and cause the *mksf*(1M) command to fail.

Density codes listed in **<sys/mtio.h>** have device-dependent behaviors. See the hardware manual for your tape device to find which densities are valid. For some devices, these values may be referred to as formats instead of densities.

Use of unbuffered mode can reduce performance and increase media wear.

DEPENDENCIES

Driver-Specific Options for stape (Major Number 205)

The following options may be used in creating legacy DSFs for tape drives that access the **stape** driver:

e Exhaustive mode is enabled (default is disabled).

When exhaustive mode is enabled, the driver will, if necessary, attempt several different configuration options when opening a device. The first attempt follows the minor number configuration exactly, but if that fails, the driver attempts other likely configuration values.

With Exhaustive mode disabled, the driver makes only one attempt to configure a device using the configuration indicated in the minor number.

p Specifies a partitioned tape whose currently active partition is partition 1 (closest to BOT (beginning of tape)). Optional partition 1 is closest to BOT for possible use as a volume directory. The default partition without this option is partition 0. If partitioning is unsupported, the entire tape is referred to as partition 0.

s[#] Specifies fixed-block mode; the optional number indicates the block size. If the number is not present, the driver selects a default block size appropriate to the device type.

m

AUTHOR

mt was developed by HP and the University of California, Berkeley.

FILES

/dev/rtape/*	Persistent tape DSFs claimed by the estape driver
/dev/rmt/*	Legacy tape DSFs
<sys/mtio.h>	Constants and macros for use with tapes
/etc/mtconfig	Configuration property table for tapes
/dev/rmt/*config	Device files for accessing configuration properties table - for internal use only

SEE ALSO

dd(1), mt(1), insf(1M), ioscan(1M), lssf(1M), mksf(1M), rmsf(1M), ioctl(2), lseek(2), libIO(3X), intro(7).

Configuring HP-UX for Peripherals

NAME

ndp - Neighbor Discovery Protocol, NDP

DESCRIPTION

Neighbor Discovery Protocol (NDP) is a protocol used by hosts and routers to:

1. Find the link-layer address of the neighbors known to be attached to the same link.
2. Find the neighboring routers that are willing to forward packets on their behalf.
3. Actively keep track of which neighbors are reachable and which are not.
4. Search for alternate routers when the path to a router fails.

To accomplish the above mentioned tasks, NDP defines the following processes:

1. Router and Prefix Discovery

Router discovery is a process through which hosts locate the neighboring routers and learn prefix plus other parameters necessary for address autoconfiguration.

Prefix discovery is used by the hosts to learn the range of IPv6 addresses that reside on-link and can be reached without going through a router.

Routers send Router Advertisements which will make the hosts treat them as the default routers. The Router Advertisements will also contain prefix information options that will identify the range of IPv6 addresses that are on-link (Subnet prefix).

2. Router and Host Requirements

Router requirements in NDP specify a set of rules for host to act as a router. These rules include:

- Router configuration variables.

  These configuration variables include intervals between successive unsolicited router advertisements, etc.

- How to make an interface an advertising interface.

  When an interface is made an advertising interface, it means that the node is going to send periodic router advertisements and is willing to forward packets on behalf of hosts on that link.

- Message content for router advertisements.

  A router will send periodic as well as solicited Router Advertisements on an advertising interface. NDP specifies the format of these messages.

- Sending unsolicited router advertisements.

  Apart from sending solicited router advertisements in response to router solicitations, routers can send unsolicited router advertisements. For example, unsolicited router advertisements can be sent to expire a prefix or to advertise a new prefix, etc.

- Stopping router advertisements on an interface.

  A router can stop advertising prefixes on an interface. This can happen due to system management decisions when a router may be stopped from being one. NDP specifies what the router should be doing under these circumstances.

- Processing router solicitation messages.

  Hosts as part of the stateless autoconfiguration process will send Router Solicitations. Routers should respond to such solicitations with a router advertisement.

- Steps to be taken when the link-local address for the router changes.

  Normally the link-local address of a Router should not change. However, NDP still defines the steps should be taken by the router when its link-local address changes for any of its interfaces.

Host requirements are a set of rules that apply for a IPv6 host. They are:

- IPv6 variables that have to be maintained.

  These variables include the time between retransmissions of neighbor solicitations, link MTU for each interface, etc.

- Processing router advertisements.

  This rule discusses what actions should be taken on receipt of router advertisements.

- Timing out prefixes and default routers.

  Whenever routers send router advertisements, they include the lifetime of the router as well as the prefixes that they advertise. NDP specifies what actions the host should take when these lifetimes expire.

- Selecting a default router.

  When there is more than one router in the link, the default router selection algorithm comes into effect. This algorithm helps select the default router based on factors like reachability, etc.

- Sending a router solicitation.

  When an interface is enabled, a host need not wait for the unsolicited router advertisement. Instead, it can send a router solicitation and get a router advertisement as a response. This will help in receiving the default router and prefix information as soon as the interface is enabled.

3. Algorithm for Sending a Packet

Any IPv6 host is required to maintain some data structures that will be used by the algorithm for sending a packet. These data structures are:

Neighbor Cache
A set of entries that will maintain IPv6 Address to link-layer address mappings for neighbors to which a packet has been sent recently. In addition to that it maintains information needed for neighbor unreachability detection like the reachability state, etc.

Destination Cache
A set of entries for hosts to whom packets have been sent recently. This includes hosts which are both on-link and off-link. It contains a level of indirection to the neighbor cache.

Prefix List
This is a list of prefixes which define the set of IPv6 address that are on-link. This information is maintained on a per interface basis. Typically this list is built from Router Advertisements received from the router.

Default Router List
A list of routers which will forward packets on behalf of this host. This list will again have a pointer to a neighbor cache entry for the respective router.

A host will use the above data structures while sending a packet to a host. Following is the conceptual algorithm for sending a packet to a unicast destination.

a. Before a packet is sent out, the next hop should be determined. Normally, next hop determination is not done on all packets. The results of a next hop determination are stored in the destination cache. The host should first check the destination cache for any entry that matches with the current destination address. If it finds a match, then it proceeds to step c, below.

b. If there is no entry for the destination in the destination cache, a longest prefix match is made with all prefixes in the prefix list. If there is a match, the destination is determined to be on-link and the destination address will be considered as the next hop. Otherwise, the next hop is determined from the routing table.

c. Once the next hop is determined, the address resolution process and neighbor unreachability detection are done for the next hop. This process is explained in the next section.

d. Once the neighbor is known to be reachable, the packet is sent to that destination.

4. Address Resolution and Neighbor Unreachability Detection

Address resolution is a process used to determine the link-layer address of a neighbor. The IPv6 Address to link-layer address mapping found through this process is cached in the Neighbor Cache. Following are the steps involved in Address Resolution.

a. First, the neighbor cache is checked for an entry which matches the current destination address. If the entry is not present, the host sends a Neighbor Solicitation Message to the solicited-node multicast group. This multicast address is derived based on the destination IPv6 address and all nodes with the particular IPv6 address are required to join that group.

b. If a host with the specified IPv6 address is present in the network, it will reply this solicitation with a Neighbor Advertisement Message.

c. On receiving the Neighbor Advertisement, the node will search for an entry in the neighbor cache for the sender's IPv6 address. A new entry is created in the neighbor cache and the reachability flag is set to REACHABLE.

Once the Address resolution is completed, neighbor unreachability detection will be performed. This process depends on the reachability field of the neighbor cache. An entry in the neighbor cache can have any of the following states:

INCOMPLETE
: The address resolution is in progress and the link-layer address of the destination is yet to be determined.

REACHABLE
: The destination is reachable until recently.

STALE
: The destination is no longer known to be reachable, but reachability detection need not be made until a packet has to be sent to that destination.

DELAY
: This state is an optimization that gives additional time for the upper layer protocols to provide the reachability confirmation.

PROBE
: A reachability confirmation is actively requested by repeatedly sending Neighbor Solicitations.

During neighbor unreachability detection, the node checks for the state in the neighbor cache. If the state for the destination is REACHABLE, the packet is sent. Otherwise, the following steps are taken:

a. When an address resolution is made on a destination, an entry is created in the neighbor cache for that destination and the reachability state will be set to INCOMPLETE. If the address resolution fails, the entry is deleted.

b. When the address resolution passes, the entry will be filled with the destination's link-layer address and the state will be set to REACHABLE.

c. There is a timer maintained called the Reachability timer which will expire the state of an entry in the neighbor cache. Once this timer expires, the reachability state changes from REACHABLE to STALE.

d. When a packet is being sent to a destination whose state is STALE in the neighbor cache, the node sets the state to DELAY and starts a timer associated with that state. By the time the timer expires if the node received reachability confirmation, the state is set to REACHABLE. Otherwise, it is set to PROBE.

e. Once the entry's state is in PROBE, the node sends unicast neighbor solicitations to the link-layer address specified in the entry. If it receives a neighbor advertisement in response the state is set to REACHABLE. This solicitation will be sent repeatedly; the maximum number of times is configurable. If the reachability confirmation is not received after maximum solicitations, the entry is deleted from the neighbor cache and the address resolution is done again.

Note: Entries in the neighbor cache can also be created as a result of node receiving unsolicited Neighbor Advertisements, Router Advertisements and Router Solicitations, etc. However, for the entry created under these circumstances the reachability state will always be set to STALE.

5. Redirect Function

A router will send a host a redirect message when it finds that there is a better next-hop router on the same link. This is a requirement for a router.

On receiving a router redirect message, a host should update its destination cache with the new next hop address.

AUTHOR

NDP was developed by the IPng Working Group of the Internet Engineering Task Force.

SEE ALSO

ifconfig(1M), ndp(1M), IPv6(7P), lan(7).

n

Neighbor Discovery for IPv6, RFC2461, T. Narten et al. NDP Neighbor Discovery Protocol

NAME

nfs, NFS - network file system

DESCRIPTION

The Network File System (NFS) allows a client node to perform transparent file access over the network. By using NFS, a client node operates on files residing on a variety of servers and server architectures, and across a variety of operating systems. File access calls on the client (such as read requests) are converted to NFS protocol requests and sent to the server system over the network. The server receives the request, performs the actual file system operation, and sends a response back to the client.

NFS operates in a stateless manner using remote procedure calls (RPC) built on top of an external data representation (XDR) protocol. The RPC protocol enables version and authentication parameters to be exchanged for security over the network.

A server grants access to a specific file system to clients by adding an entry for that file system to the server's **/etc/dfs/dfstab** file.

A client gains access to that file system using the **mount** command to request a file handle for the file system (see *mount*(1M)). (A file handle is the means by which NFS identifies remote files.) Once a client mounts the file system, the server issues a file handle to the client for each file (or directory) the client accesses. If the file is removed on the server side, the file handle becomes stale (dissociated with a known file), and the server returns an error with **errno** set to [ESTALE].

A server can also be a client with respect to file systems it has mounted over the network; however, its clients cannot directly access those file systems. If a client attempts to mount a file system for which the server is an NFS client, the server returns with **errno** set to [EREMOTE]. The client must mount the file system directly from the server on which the file system resides.

The user ID and group ID mappings must be the same between client and server. However, the server maps UID 0 (the superuser) to UID –2 before performing access checks for a client. This process prevents gaining superuser privileges on remote file systems.

RETURN VALUE

Generally, physical disk I/O errors detected at the server are returned to the client for action. If the server is down or inaccessible, the client receives the message:

```
NFS: file server xxx not responding: still trying.
```

where **xxx** is the hostname of the NFS server. The client continues resending the request until it receives an acknowledgement from the server. Therefore, the server can crash or power down, and come back up without any special action required by the client. The client process requesting the I/O will block, but remains sensitive to signals (unless mounted with the **nointr** option) until the server recovers. However, if mounted with the **soft** option, the client process returns an error instead of waiting indefinitely.

AUTHOR

nfs was developed by Sun Microsystems, Inc.

SEE ALSO

exportfs(1M), share(1M), mount(1M), mount_nfs(1M), nfsd(1M), mount(2), fstab(4), dfstab(4).

NAME

null - null file

DESCRIPTION

Data written on a null special file is discarded.

Reads from a null special file always return 0 bytes.

EXAMPLES

To create a zero-length file, use either of the following:

cat /dev/null > *file*
cp /dev/null *file*

FILES

/dev/null

STANDARDS CONFORMANCE

null: AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, POSIX.2

NAME

pckt - Packet Mode module for STREAMS pty (pseudo-terminal)

SYNOPSIS

```
#include <sys/stropts.h>

int ioctl(fd_slave, I_PUSH, "pckt");
```

DESCRIPTION

The **Packet Mode** feature for STREAMS pty devices allows the user process on the master side of the pty device to be informed of state changes in the pty. To enable **Packet Mode** in the STREAMS pty device, the user process must push the **pckt** module onto the master side of the pty with a call to the STREAMS **I_PUSH** *ioctl*(2) system call. When the **pckt** module is pushed onto a STREAMS pty master, certain STREAMS messages going upstream on the master side will get packetized so they can be subsequently retrieved by the master side with a **getmsg** function.

When the user process writes data, the **pckt** module passes the message unchanged downstream on to the next module or driver. When the user process reads data or when the **pckt** module receives certain STREAMS message types, it constructs a packet out of the message for forwarding upstream. To construct a message packet, the module creates an **M_PROTO** message. This **M_PROTO** message contains the original message type in the first data block and the original message in as many data blocks as needed. The user process can then retrieve the **M_PROTO** message with a call to the **getmsg()** function.

The **pckt** module packetizes the following STREAMS message types:

M_DATA, **M_IOCTL**, **M_PROTO**, **M_PCPROTO**, **M_FLUSH**, **M_START**, **M_STOP**, **M_STARTI**, **M_STOPI**, **M_READ**.

All other messages are passed unchanged upstream.

If the message is an **M_FLUSH** message, the **pckt** module looks at the flag and takes the following actions:

- If the flag is **FLUSHW**, the module changes it to **FLUSHR** before creating the **M_PROTO** message and passing the message upstream. This prevents the stream head's read queue from being flushed by the original **M_FLUSH** message.
- If the flag is **FLUSHR**, the module changes it to **FLUSHW** before creating the **M_PROTO** message and passing it upstream. To flush the write queues properly, the module also sends an **M_FLUSH** message with the **FLUSHW** flag set.
- If the flag is **FLUSHRW**, the module changes it to **FLUSHW** before creating the **M_PROTO** message and passing it upstream. To flush the write queues properly, the module also sends an **M_FLUSH** message with the **FLUSHW** flag set.

AUTHOR

pckt(7) was developed by HP and OSF.

SEE ALSO

getmsg(2), ioctl(2), ptm(7), pts(7), ldterm(7), ptem(7), streamio(7).

NAME

poll - monitor I/O conditions on multiple file descriptors

SYNOPSIS

```
#include <sys/devpoll.h> #include <fcntl.h>

int open("/dev/poll", O_RDWR);

int write(int filedes, const struct pollfd *buf, size_t nbyte);

int ioctl(int filedes, DP_POLL, struct dvpoll *arg);

int ioctl(int filedes, DP_ISPOLLED, struct pollfd *arg);
```

DESCRIPTION

/dev/poll provides an interface to the event port driver allowing a user to synchronously monitor a specific set of conditions associated with a registered set of file descriptors. Poll conditions include the ability to read or write data without blocking and certain exceptional conditions.

Access to **/dev/poll** is provided through the **open()**, **write()**, and **ioctl()** system calls.

The **/dev/poll** event port provides functionality comparable to the **select(2)** and **poll(2)** system calls and supports the following types of file descriptors: network (**AF_INET**) and Unix Domain (**AF_UNIX**) sockets, named FIFO files and pipes, XTI endpoints, and STREAMS devices.

General operations supported by the event port driver are:
-- Opening an event port.
-- Registering and deregistering file descriptors on an event port.
-- Polling registered file descriptors on an event port.
-- Retrieving registered poll conditions for a file descriptor.
-- Closing an event port.

Opening An Event Port

Each open of the **/dev/poll** device enables an event port from which a different set of file descriptors can be polled. The file descriptor returned by the **open()** system call represents the event port. Users wishing to monitor multiple sets of file descriptors should open the **/dev/poll** device multiple times. For example:

```
int evpfd;

evpfd = open("/dev/poll", O_RDWR);
```

Only the process that performed the **open()** on **/dev/poll** can perform general event port operations. Specifically, any event port file descriptor inherited by a child from its parent or that is received from another process using the Unix Domain Sockets access rights can only be closed. (See *sendmsg* in the *send*(2) man page or the STREAMS **I_FDINSERT** ioctl *request* in the *streamio*(7) man page.)

Registering and Deregistering File Descriptors

An interest set of file descriptors and poll conditions is registered with an event port by using the **write()** system call. By writing an array of **pollfd** structures to an event port the user can register multiple file descriptors in one **write()** service call. The **pollfd** structure and related poll conditions are defined in **<poll.h>**, (included by **<sys/devpoll.h>**). Other flags are defined in the **<sys/devpoll.h>** file. See the *poll*(2) man page for the definition of the poll conditions.

To register a file descriptor, the **fd** field is set to the file descriptor to be registered, and the **events** field is set to one or more poll conditions, such as **POLLIN**. Multiple poll conditions can be **OR**ed together. A given file descriptor can be registered with multiple event ports. Re-registering a file descriptor with the same event port will cause the the specified poll conditions to join the previous conditions for the given file descriptor.

To deregister, **fd** is set to the file descriptor to be deregistered, and **events** is set to **POLLREMOVE**. **POLLREMOVE** is defined in **<sys/devpoll.h>**. **POLLREMOVE** must not be **OR**ed together with any other poll conditions.

When a polled file descriptor is closed, it is automatically deregistered.

Continuing our example, the following registers two file descriptors on the opened event port, **fd1** and **fd2**:

```
struct pollfd pfd[2];
int err;

pfd[0].fd = fd1;
pfd[0].events = POLLIN;
pfd[1].fd = fd2;
pfd[1].events = (POLLIN | POLLRDBAND);
err = write(evpfd, pfd, sizeof(pfd));
```

Polling File Descriptors

Polling an event port's interest set is initiated by calling **ioctl()** specifying the **DP_POLL** *request*.

The ioctl *arg* parameter is a pointer to a **dvpoll** structure, defined in **<sys/devpoll.h>**. It contains the following members:

```
struct dvpoll {
   pollfd_t *dp_fds;    /* pollfd[] to be used */
   nfds_t    dp_nfds;   /* number of pollfd entries */
   int       dp_timeout; /* milliseconds or -1 */
}
```

dp_fds is a pointer to an array of **pollfd** structures. **dp_nfds** is the maximum number of **pollfd** structures to be returned in that array. **dp_timeout** is the maximum time, in milliseconds, to wait for at least one of the registered poll conditions to be met in the event port.

When one or more registered poll conditions are met for any of the registered file descriptors, **ioctl()** stores the valid poll conditions in the **revents** of each **pollfd** structure in the array, one array element for each active file descriptor. The return value of **ioctl()** is the number of valid **pollfd** structures.

If no poll conditions are met and if **dp_timeout** is **-1**, **ioctl()** sleeps until a poll condition is met on any of the registered file descriptors. If **dp_timeout** is non-negative, **ioctl()** returns after *dp_timeout* milliseconds expires or when a poll condition is met. If the time limit expires, the **ioctl()** return value is **0**.

Retrieving Registered Poll Conditions for a File Descriptor

The registered poll conditions for a given file descriptor in an interest set can be determined by calling **ioctl()** with the **DP_ISPOLLED** *request*. For example, for file descriptor **fd1**:

```
struct pollfd pfd;
int ispolled;

pfd.fd = fd1;
ispolled = ioctl(evpfd, DP_ISPOLLED, &pfd);
```

If the file descriptor is registered with the event port, the **ioctl()** return value is **1**, and the registered poll conditions are returned in the **events** member of the **pollfd** structure.

The **ioctl()** return value is **0** if the file descriptor is not registered or is not open.

Closing an Event Port

An event port is closed with the **close()** system call specifying the event port file descriptor. All file descriptors registered with that event port are automatically deregistered from that event port.

RETURN VALUES

open() returns the event port file descriptor. If the **open()** system call fails, it returns **-1**, and **errno** is set to the error condition.

write() returns the number of bytes in the array of the **pollfd** structure that was passed in *buf*. If the **write()** returns **-1**, **errno** is set to the error condition.

ioctl(DP_POLL) returns the number of file descriptors for which one or more poll conditions are met. **ioctl(DP_POLL)** returns **0** if a timeout occurred before any poll conditions were satisfied for any of the registered file descriptors.

ioctl(DP_ISPOLLED) returns **1** if the file descriptor specified in the **pollfd** structure is registered. **ioctl(DP_ISPOLLED)** returns **0** if the file descriptor is not registered or is closed.

p

If **ioctl()** returns **-1**, **errno** is set to the error condition.

ERRORS

The following errors are returned by the event port driver.

If **open()** fails, **errno** is set to one of the following values.

[EACCES] The minor number of the device file name passed to **open()** is not **0**.

[EAGAIN] Allocation of internal data structures failed due to a temporary condition. Calling **open()** again might succeed.

[EMFILE] The maximum number of file descriptors allowed for the process is already open.

[ENFILE] The maximum number of files allowed for the system is already open.

[ENXIO] Some of the requisite file types are not supported by the **/dev/poll** driver. See the *WARNINGS* section below.

If **write()** or **ioctl()** fails, **errno** is set to one of the following values.

[EACCES] The calling process did not open the event port.

[EBADF] The *filedes* argument passed to **write()** is not an open file descriptor.

[EFAULT] An attempt was made to access a **pollfd** structure whose location is outside the process address space.

[EINTR] A signal interrupted the **ioctl(DP_POLL)** system call.

[EINVAL] The *nbyte* argument passed to **write()** is less than **0**.

[ENODEV] The *filedes* argument passed to **write()** is not an event port file descriptor.

EXAMPLES

The following examples show how to use the **/dev/poll** driver to poll for events on network socket file descriptors.

To register a TCP socket file descriptor (**sd**) so that **ioctl(DP_POLL)** will notify the application when a new connection is established or when input data is available:

```
struct pollfd regpfd;
int err;

regpfd.fd = sd;
regpfd.events = POLLIN;
err = write(evpfd, &regpfd, sizeof(regpfd));
```

POLLRDBAND should be **OR**ed with **POLLIN** if the application needs to distinguish the arrival of out-of-band data.

To wait for events on one or more registered sockets, up to 100 connections:

```
struct pollfd pollpfd[100];
struct dvpoll dvp;
int npoll;

dvp.dp_fds = pollpfd;
dvp.dp_nfds = 100;
dvp.dp_timeout = -1;
npoll = ioctl(evpfd, DP_POLL, &dvp);
```

If a non-blocking write to a socket is incomplete, the following can be used to register the socket so that **ioctl(DP_POLL)** will notify the application when the socket is writable again later. Typically, the socket is already registered to receive input notifications. The following will add the **POLLOUT** notification.

```
struct pollfd regpfd;
int err;

regpfd.fd = sd;
regpfd.events = POLLOUT;
err = write(evpfd, &regpfd, sizeof(regpfd));
```

p

After the last non-blocking write succeeds, the following should be used to deregister for **POLLOUT**, but continue to be registered for input notifications. Note that **POLLREMOVE** must be used in order to remove the **POLLOUT** registration.

```
struct pollfd regpfd[2];
int err;

regpfd[0].fd = sd;
regpfd[0].events = POLLREMOVE;
regpfd[1].fd = sd;
regpfd[1].events = POLLIN;
err = write(evpfd, regpfd, sizeof(regpfd));
```

The following uses **ioctl(DP_ISPOLLED)** to demonstrate how to accomplish the same thing in the more general case, for example, when an application library might not know how the file descriptor is normally registered.

```
struct pollfd regpfd[2];
int err;

regpfd[0].fd = sd;
regpfd[0].events = POLLREMOVE;
regpfd[1].fd = sd;
err = ioctl(evpfd, DP_ISPOLLED, &regpfd[1]);
regpfd[1].events &= ~POLLOUT;    /* clear POLLOUT */
err = write(evpfd, regpfd, sizeof(regpfd));
```

WARNINGS

/dev/poll usually performs better than **select()** and **poll()** especially when the application has registered a very large number of file descriptors. However, in cases where specified conditions are likely to occur simultaneously on a large number of registered file descriptors, performance levels will be diminished.

If **open()** returns **-1** and **errno** is set to [ENXIO], this indicates that some of the necessary system patches have not been installed, and the system administrator must install the File System, Transport, and STREAMS patches that support **/dev/poll** (event ports).

The **write()** system call does not return any error indication if one or more of the file descriptors in the **pollfd** structure could not be registered or deregistered.

If **POLLREMOVE** is **OR**ed with other poll conditions in a **pollfd** structure passed to **write()**, **POLLREMOVE** is ignored. The other poll conditions will be **OR**ed with any existing poll conditions for the registered file descriptor.

The **ioctl(DP_POLL)** system call returns only the first *dp_nfds* active file descriptors. There is no indication if there are additional active file descriptors.

The **ioctl(DP_ISPOLLED)** system call also returns its result in the **revents** member of the **pollfd** structure, in order to be compatible with the implementation of the **/dev/poll** driver by some other vendors.

The **ioctl(DP_ISPOLLED)** system call does not return any error indication if the file descriptor in the **pollfd** structure is not open.

When an event port is closed, the **close()** system call might take a noticeable amount of time to complete if a very large number of file descriptors is still registered.

AUTHOR

The event port driver was developed independently by HP.

FILES

/dev/poll	driver device file
/sbin/init.d/devpoll	start-up script that creates **/dev/poll**
/etc/rc.config.d/devpoll	configuration parameters for start-up script

SEE ALSO

ioctl(2), mknod(2), open(2), pipe(2), poll(2), select(2), send(2), socket(2), socketpair(2), write(2), t_open(3).

NAME

ps2, ps2kbd, ps2mouse - PS/2 keyboard/mouse device driver and files

SYNOPSIS

#include <sys/ps2io.h>

DESCRIPTION

The **ps2** driver allows the use of IBM Personal System/2 (PS/2) compatible keyboards and mouse devices on Hewlett-Packard workstations equipped with PS/2 interface hardware.

On systems with a single interface, PS/2 device file names use the following format:

/dev/ps2_*n*

where *n* represents the interface port number, ranging from 0 to 15. For example, the device file **/dev/ps2_1** is used to access port one.

On systems with more than one interface, PS/2 device file names use the following format:

/dev/ps2_*m***.***n*

where *m* represents the interface number, and *n* represents the port number. For example, the device file **/dev/ps2_1.2** is used to access port two on interface one.

At boot time, the **ps2** driver scans all interface ports from port zero to the maximum number of ports implemented and attempts to identify attached PS/2 devices. The **/dev/ps2mouse** device file accesses the first mouse detected by **ps2**. The **/dev/ps2kbd** device file accesses the first keyboard detected by **ps2**.

PS/2 devices are classified as "slow" devices. This means that system calls to **ps2** can be interrupted by caught signals (see *signal*(5)).

The mouse may be placed in one of two output modes. In stream mode, the mouse generates a three-byte report packet in response to mouse movement and/or button presses. These reports can be obtained with the **read()** system call (see *read*(2)). In prompt mode, an **ioctl()** request polls the mouse, returning a three-byte report packet in a buffer whose address is passed as an argument to the **ioctl()** call.

PS/2 keyboards return keycodes that represent key-press and key-release events. Use the Internal Terminal Emulator (ITE) to read ASCII characters from PS/2 keyboards. The ASCII terminal interface used by the ITE is described in *termio*(7).

The **ps2** driver provides a low-level programming interface to PS/2 keyboards and mice. To access these devices in a hardware independent way, use the X Window programming environment.

System Calls

The **open()** system call gives exclusive access to the specified PS/2 device (see *open*(2)). If a port is open, all **open()** calls made on that port will fail with **errno** set to [EBUSY] (see *errno*(2)).

If an open is attempted on a nonexistent port, the **open()** call fails with **errno** set to [ENXIO].

If no keyboard is detected at system boot and an **open()** is attempted on **/dev/ps2kbd**, or if no mouse is detected at system boot and an **open()** is attempted on **/dev/ps2mouse**, the **open()** call fails with **errno** set to [ENXIO].

Attempts to open an existing **ps2** port with no device connected will succeed.

Upon a successful open, any previously queued input from the device is discarded. Keystrokes are routed to the ITE by default. While a keyboard is open, ITE does not receive keystrokes from that keyboard; until the keyboard device is closed, it has exclusive access to keyboard input.

The file status flags **O_NDELAY** and **O_NONBLOCK** can be set to enable nonblocking reads (see *open*(2)).

read() returns bytes from a PS/2 device. HP-UX maintains a 512-byte buffer for each port. When this buffer is full, additional bytes received from the device are discarded.

If enough buffered data is available to satisfy the entire number of bytes requested, the **read()** call completes successfully, having read all of the data requested and returning the number of bytes read.

If there is not enough buffered data available to satisfy the entire request, but at least one byte is available, the **read()** call completes successfully, having read all available data and returning the number of bytes actually read.

If both file status flags **O_NDELAY** and **O_NONBLOCK** are clear and no data is available, the **read()** call blocks until data becomes available or a signal is received.

If the file status flag **O_NDELAY** is set and no data is available, the **read()** call returns zero instead of blocking.

If the file status flag **O_NONBLOCK** is set and no data is available, the **read()** call returns **-1** with **errno** set to [EAGAIN] (see *errno*(2)).

The **write()** system call is not supported by **ps2**.

The **select()** system call can be used to determine if data is currently available to be read from a **ps2** port. Using **select()** for write or for exception conditions always returns a false indication in the file descriptor bit masks (see *select*(2)).

The **ioctl()** system call is used to perform special operations on PS/2 mouse and keyboard devices (see *ioctl*(2)). The set of **ps2** driver **ioctl()** requests are divided into three groups: general requests to both mouse and keyboard, keyboard-specific requests, and mouse-specific requests. Mouse-specific requests used on keyboards, and keyboard-specific requests used on mice, fail, returning **-1** with **errno** set to [EINVAL].

Any **ioctl()** request (except **PS2_PORTSTAT**) used on a port not connected to a PS/2 device will time out, returning **-1** with **errno** set to [EIO].

All **ioctl()** system calls use the following syntax:

```
int ioctl(int fildes, int request, char *arg);
```

All requests that require parameters or return data use a 4-byte unsigned character buffer addressed by the *arg* argument.

The *request* codes that follow are defined in **<sys/ps2io.h>**.

General ioctl() Requests for Both Keyboard and Mouse

PS2_PORTSTAT
: Return driver status information.

Two bytes of data are returned in the character buffer addressed by *arg*.

Byte 0, which indicates the type of connected device, can have four possible values:

PS2_NONE	No device is detected.
PS2_MOUSE	Mouse is detected.
PS2_KEYBD	Keyboard is detected.
PS2_UNKNOWN	Unknown device is detected.

Byte 1 contains bit flags for various pieces of driver information. The following bit masks for this byte are defined in the file **/usr/include/sys/ps2io.h**:

INTERFACE_HAS_ITE	If set, the interface containing this port is used by the Internal Terminal Emulator (ITE) for keyboard input.
PORT_HAS_FIRST_KEYBD	If set, this port is connected to the first keyboard detected by the driver.
PORT_HAS_FIRST_MOUSE	If set, this port is connected to the first mouse detected by the driver.

All other bits are currently unused, and are cleared to zero.

PS2_DISABLE
: Disable a PS/2 device.

Further output from the device is prevented by the device itself. This request does not use *arg*. Certain devices perform actions in addition to disabling themselves.

The keyboard resets its internal state to the default state, stops scanning the keys, and waits for further commands.

The mouse stops transmission of reports, and then disables itself.

p

PS2_ENABLE Enable a PS/2 device

Transmissions from the device are enabled. This request does not use *arg*.

PS2_IDENT Identify a PS/2 device.

A value identifying the type of device is returned in the 4-byte buffer addressed by *arg*. The keyboard returns two bytes (*arg* **[0]=0xAB** and *arg* **[1]=0x83**). The mouse returns one byte (*arg* **[0]=0x00**).

PS2_SETDEFAULT Set the device to its default (power-up) state.

The device is returned to its default internal state. This request does not use *arg*.

PS2_RESET Reset a PS/2 device.

The device is told to execute its internal reset routine and execute its power-up test. The result of the power-up test is returned in the 4-byte buffer addressed by *arg*. The mouse returns two bytes to indicate a successful reset (*arg* **[0]=0xAA** and *arg* **[1]=0x00**). The keyboard returns one byte (*arg* **[0]=0xAA**).

Keyboard-Specific ioctl() Requests

PS2_SCANCODE Select the keyboard scancode set

The scancode set to be used by the keyboard is passed as the first byte of the buffer addressed by *arg*. The following are valid values for this byte:

SCANCODE_1	Selects scancode set 1.
SCANCODE_2	Selects scancode set 2.
SCANCODE_3	Selects scancode set 3.
GET_SCANCODE	Returns the scancode used.

When **GET_SCANCODE** is specified, the scancode used by the keyboard is returned as the first byte of the character buffer addressed by *arg*. Some keyboards do not support all scancode sets.

PS2_ALL_TMAT Set all keys to typematic behavior.

This request can be made when the keyboard is using any scancode set; however, it affects only the operation of scancode set 3. The *arg* parameter is not used. The typematic rate and delay are set via the **PS2_RATEDELAY ioctl()** request.

PS2_ALL_MK Set all keys to make-only behavior.

This request can be made when the keyboard is using any scancode set; however, it affects only the operation of scancode set 3. The *arg* parameter is not used.

PS2_ALL_MKBRK Set all keys to make/break behavior.

This request can be made when the keyboard is using any scancode set; however, it affects only the operation of scancode set 3. The *arg* parameter is not used.

PS2_ALL_TMAT_MKBRK Set all keys to typematic make/break behavior.

This request can be made when the keyboard is using any scancode set; however, it affects only the operation of scancode set 3. The *arg* parameter is not used. The typematic rate and delay are set via the **PS2_RATEDELAY ioctl()** request.

PS2_KEY_TMAT Set typematic behavior for an individual key.

The key code from scancode set 3 for the individual key is passed as the first byte in the character buffer addressed by *arg*. This request can be made when the keyboard is using any scancode set; however, it affects only the operation of scancode set 3. The typematic rate and delay are set via the **PS2_RATEDELAY ioctl()** request. Because keyboards might be left in a disabled state after this request, the **PS2_ENABLE** request should be performed after **PS2_KEY_TMAT**.

PS2_KEY_MAKE Set make-only behavior for an individual key.

The key code from scancode set 3 for the individual key is passed as the first byte in the character buffer addressed by *arg*. This request can be made when the keyboard is using any scancode set; however, it affects only the operation of scancode set 3. Because keyboards might be left in a disabled state after this request, the **PS2_ENABLE** request should be performed after **PS2_KEY_MAKE**.

PS2_KEY_MKBRK Set make/break for an individual key.

The key code from scancode set 3 for the individual key is passed as the first byte in the character buffer addressed by *arg*. Make/break behavior will be set for this key. This request can be made when the keyboard is using any scancode set; however, it affects only the operation of scancode set 3. Because keyboards might be left in a disabled state after this request, the **PS2_ENABLE** request should be performed after **PS2_KEY_MKBRK**.

PS2_INDICATORS Set the state of keyboard indicators, Num Lock, Caps Lock, and Scroll Lock, according to the value passed in the first byte of the character buffer addressed by *arg*.

The indicators are bit-mapped as follows:

NONE_LED	No indicators active
CAPS_LED	Caps Lock indicator active
NUM_LED	Num Lock indicator active
SCROLL_LED	Scroll Lock indicator active

PS2_RATEDELAY Set the rate and delay for all typematic keys by specifying the value passed as the first byte in the character buffer addressed by *arg*.

Bits zero through four give the rate. Bits five and six give the delay. Bit seven (the most significant bit) is unused and should be set to zero. The delay in milliseconds is determined by the following equation, where *X* is the numeric value of bits five through six:

delay **= (1+*X*) * 250** *(+|−20%)*

The period (interval from one output key code to the next) in seconds is determined by the following equation, where *Y* is the numeric value of bits zero through two, and *Z* is the numeric value of bits three through four:

period **= (8+*Y*) * (2^*Z*) * 0.00417** *(+|−20%)*

The typematic rate (expressed in make codes per second) is one for each period using the above equation. The default typematic rate is 10.9 characters per second. The default delay is 500 milliseconds.

Mouse-Specific ioctl() Requests

PS2_SAMPLERATE Set the mouse sampling rate used in stream mode by specifying the value passed as the first byte in the character buffer addressed by *arg*.

Seven specific rates are supported:

SAMPLE_10	10 reports/second maximum
SAMPLE_20	20 reports/second maximum
SAMPLE_40	40 reports/second maximum
SAMPLE_60	60 reports/second maximum
SAMPLE_80	80 reports/second maximum
SAMPLE_100	100 reports/second maximum
SAMPLE_200	200 reports/second maximum

The default rate is 100 reports/second maximum. This request updates the mouse sampling rate only in stream mode. If the mouse is in prompt mode, this request is ignored.

PS2_PROMPTMODE Put mouse into prompt mode.

p

In prompt mode, the mouse updates its internal values due to movement or button presses, but issues reports only in response to the **PS2_REPORT** **ioctl()** request. The *arg* parameter is not used.

PS2_REPORT Obtain a prompt mode mouse report.

This request polls the mouse, obtaining a three-byte report returned in the character buffer addressed by the *arg* parameter. The report has the following format:

Byte 1 A bit map of buttons, signs, and overflows

Bit 0	Left button (1=depressed)
Bit 1	Right button (1=depressed)
Bit 2	Center button (1=depressed)
Bit 3	Always 1
Bit 4	X data sign (1=negative)
Bit 5	Y data sign (1=negative)
Bit 6	X data overflow (1=overflow)
Bit 7	Y data overflow (1=overflow)

Byte 2 X-coordinate data byte

Byte 3 Y-coordinate data byte

The X and Y coordinate values are expressed in two's complement. The scaling behavior specified via the **PS2_2TO1_SCALING ioctl()** request does not apply to reports obtained with the **PS2_REPORT ioctl()** request. **PS2_2TO1_SCALING** affects only reports sent in stream mode.

PS2_STREAMMODE Put mouse into stream mode.

When in stream mode, the mouse sends a three-byte report whenever the mouse is moved, or a button is pressed or released since the last report. The maximum report rate is set with the **PS2_SAMPLERATE ioctl()** request. If a button is both pressed and then released within a sample interval, it will be reported as pressed at the end of that interval.

The stream-mode reports are obtained via the **read()** system call (see *read*(2)). The format of the report is identical to reports returned by the **PS2_REPORT ioctl()** request described above.

When in stream mode, the **PS2_DISABLE** request must be sent prior to any other **ioctl()** requests.

The *arg* parameter is not used.

PS2_STATUS Obtain mouse status.

This request polls the mouse, obtaining a three-byte report returned in the character buffer addressed by the *arg* parameter.

The status report has the following format:

Byte 1 A bit map of buttons and mouse internal state

Bit 0	Right button (1=depressed)
Bit 1	Center button (1=depressed)
Bit 2	Left button (1=depressed)
Bit 3	Always 0
Bit 4	If 0, scaling 1:1; if 1, scaling 2:1
Bit 5	If 0, disabled; if 1, enabled
Bit 6	If 0, stream mode; if 1, prompt mode
Bit 7	Always 0

Byte 2 Current resolution setting

Byte 3 Current sampling rate

PS2_RESOLUTION Set mouse resolution for X and Y coordinate values by specifying the value passed as the first byte in the character buffer addressed by *arg*. Four discrete resolutions are supported:

p

Resolution	200 DPI	320 DPI
RES_1	1 count/mm	1 count/mm
RES_2	2 count/mm	3 count/mm
RES_3	4 count/mm	6 count/mm
RES_4	8 count/mm	12 count/mm

PS2_2TO1_SCALING Set mouse scaling at 2 to 1. The X and Y coordinate values returned in stream-mode reports are doubled, except for absolute values less than six, which are converted to new values in a nonlinear fashion. The conversion is detailed in this table:

Mouse Internal Value	Converted Value
0	0
+|− 1	+|− 1
+|− 2	+|− 1
+|− 3	+|− 3
+|− 4	+|− 6
+|− 5	+|− 9
All other *n*	2 * *n*

This conversion does not apply to reports obtained via the **PS2_REPORT** **ioctl()** request.

The *arg* parameter is not used.

PS2_1TO1_SCALING Set mouse scaling at 1 to 1.

The X and Y values returned in mouse reports are not scaled. This request does not use the *arg* parameter.

ERRORS

If a system call fails, as noted above in the DESCRIPTION section **errno** is set to one of the following values:

[EBUSY] The specified PS/2 device is already opened.

[EFAULT] A bad address was detected while attempting to use an argument to a system call.

[EINTR] A signal interrupted an **open()**, **read()**, or **ioctl()** system call.

[EINVAL] An invalid parameter was detected by **ioctl()**.

[EIO] A hardware or software error occurred while executing an **ioctl()** system call.

[ENODEV] **write()** is not implemented for PS/2 devices.

[ENXIO] No device is present at the specified address.

p

EXAMPLES

Assume that *fildes* is a valid file descriptor for a **ps2** port connected to a keyboard. The first example blinks the keyboard indicators, selects scancode set 3, and loops forever while printing keycodes.

```
#include <sys/ps2io.h>

unsigned char kbdbuf[4];  /* buffer for ioctl operations */
unsigned char inchar;     /* keycode read */

/* flash the LED indicators */
kbdbuf[0] = CAPS_LED | SCROLL_LED | NUM_LED;   /* all on */
if( ioctl( fildes, PS2_INDICATORS, &kbdbuf) < 0){
   perror("ioctl PS2_INDICATORS failed");
   exit(1);
}
printf("Indicators on\n");
sleep(1);

kbdbuf[0] = NONE_LED;  /* all off */
if( ioctl( fildes, PS2_INDICATORS, &kbdbuf) < 0){
   perror("ioctl PS2_INDICATORS failed");
```

```
      exit(1);
   }
   printf("Indicators off\n");

   /* use scancode set 3 */
   kbdbuf[0] = SCANCODE_3;
   if( ioctl( fildes, PS2_SCANCODE, &kbdbuf) < 0){
      perror("ioctl PS2_SCANCODE failed");
      exit(1);
   }

   /* identify our scancode set */
   kbdbuf[0] = GET_SCANCODE;
   if( ioctl( fildes, PS2_SCANCODE, &kbdbuf) < 0){
      perror("ioctl PS2_SCANCODE failed");
      exit(1);
   }
   printf("Keyboard reports it is using scancode set %d\n",
          (unsigned int) kbdbuf[0]);

   /* now, loop forever while printing keycodes */
   while( 1){
        read( fildes, &inchar, 1);
        printf("Keycode: %x\n", (unsigned int)inchar);
   }
```

The following example puts the mouse in stream mode, sets the report limit to 80 per second, enables the mouse, and then loops forever printing mouse reports. Assume that *fildes* is a valid file descriptor for a **ps2** port connected to a mouse.

```
   #include <sys/ps2io.h>

   unsigned char buf[3];        /* mouse report buffer */
   unsigned char ioctl_buf[4];  /* mouse ioctl buffer */

   /* first, disable the mouse */
   if (ioctl( fildes, PS2_DISABLE) < 0){
      perror("ioctl PS2_DISABLE failed\n");
      exit(1);
   }
   printf("Mouse disabled\n");

   /* Put mouse in stream mode */
   if (ioctl( fildes, PS2_STREAMMODE) < 0){
      perror("ioctl PS2_STREAMMODE failed\n");
      exit(1);
   }
   printf("Mouse in stream mode\n");

   /* set samplerate */
   ioctl_buf[0] = SAMPLE_80;
   if (ioctl( fildes, PS2_SAMPLERATE, ioctl_buf) < 0){
      perror("ioctl PS2_SAMPLERATE failed\n");
      exit(1);
   }
   printf("Mouse sample rate set to SAMPLE_80\n");

   /* Enable mouse */
   if (ioctl( fildes, PS2_ENABLE) < 0){
      perror("ioctl PS2_ENABLE failed\n");
      exit(1);
   }
   printf("Mouse enabled.\n");
```

```
for (;;) {
   if (read(fildes, &buf[0], 1) != 1){
      perror("Read of report byte 1 failed");
      return 1;
   }
   if (read(fildes, &buf[1], 1) != 1){
      perror("Read of report byte 2 failed");
      return 1;
   }
   if (read(fildes, &buf[3], 1) != 1){
      perror("Read of report byte 3 failed");
      return 1;
   }
   printf("mouse: 0x%02x, %d %d\n", buf[0], buf[1], buf[2]);
}
```

AUTHOR

ps2 was developed by the Hewlett-Packard Company.

PS/2 and Personal System/2 are registered trademarks of International Business Machines, Incorporated, in the U.S. and other countries.

FILES

/usr/include/sys/ps2io.h
/dev/ps2_[0-15]
/dev/ps2_*.[0-15]
/dev/ps2mouse
/dev/ps2kbd

SEE ALSO

close(2), errno(2), fcntl(2), ioctl(2), open(2), read(2), select(2), signal(5), termio(7).

SoftPC User's Guide

SoftPC Installation Guide

Sun System Administrators Guide for the HP700/RX

NAME

ptem - STREAMS pty (pesudo-terminal) Emulation module

SYNOPSIS

```
#include <sys/stropts.h>

int ioctl(fd_slave, I_PUSH, "ptem");
```

DESCRIPTION

ptem is a STREAMS module that emulates a terminal when used in conjunction with **ldterm** (STREAMS line discipline) and **pts** (STREAMS slave pty driver). The **ptem** module normally sits above **pts** and below **ldterm**. The user process must push the **ptem** module onto the slave side of the pty with a call to the STREAMS **I_PUSH** *ioctl*(2) system call before **ldterm** is pushed. **ptem** is responsible for processing all of the terminal **ioctl** commands that are passed downstream from **ldterm** or from **ptm** (STREAMS pty master driver).

ldterm and **ptem** together provide a real terminal behavior for the STREAMS pty slave. However, some of the terminal **ioctl** commands are ignored and cause only an acknowledgement of the command since there is no real terminal or modem in the pty subsystem. In fact, none of the flags in the **c_clfag** field of the **termio** or **termios** structures, (which is used by the **TCSETA** or **TCSETS ioctls**, respectively), have any effect on the pty except if the baud rate is set to zero. Setting the baud rate to zero will have the effect of hanging up the pty connection. Similarly, the parity or delay flags in the **c_iflag** field will not have any effect at all on the pty.

As a summary, the **ptem** module performs the following tasks:

- The following **ioctls** are processed, if appropriate, and acknowledged by sending an **M_IOCACK** message upstream when they are received on **ptem**'s write queue:

  TCSETA, **TCSETAW**, **TCSETAF**, **TCSETS**, **TCSETSW**, **TCSETSF**, **TCGETA**, **TCGETS**, and **TCSBRK**.

- Keeps track of the window size needed for the **TIOCSWINSZ**, **TIOCGWINSZ**, and **JWINSIZE ioctls**.

- Upon receiving any other **ioctl** on its write queue, **ptem** acknowledges them negatively by sending an **M_IOCNAK** message upstream.

- The following **ioctls** are passed downstream by **ptem** after they have been processed:

  TCSETA, **TCSETAW**, **TCSETAF**, **TCSETS**, **TCSETSW**, **TCSETSF**, **TCSBRK**, and **TIOCSWINSZ**.

- Any **M_IOCNAK** message that is received on **ptem**'s read queue will be freed in case the **pckt** module is not pushed on the **ptm** and the above **ioctls** get to the pty master STREAMS head, which would then send an **M_IOCNAK** message downstream.

- When **ptem** is opened and all conditions for setting up a controlling terminal are met, it sends an **M_SETOPTS** message (with the **SO_ISATTY** flag set) upstream to the STREAMS head to allocate a controlling terminal.

- Upon receiving an **M_IOCTL** message of type **TCSBRK** on its read queue, **ptem** sends an **M_IOCACK** message downstream and an **M_BREAK** message upstream.

- When an **ioctl** message is received on its write queue to set the baud rate to zero (e.g. **TCSETA** with CBAUD set to B0), **ptem** sends an **M_IOCACK** message upstream and a zero-length message downstream to be read by the pty master process.

- When an **M_IOCTL** message of type **TIOCSIGNAL** is received on its read queue, **ptem** sends an **M_IOCACK** message downstream and an **M_PCSIG** message upstream with the signal number set to the same value used in the **M_IOCTL** message.

- When an **M_IOCTL** message of type **TIOCREMOTE** is received on its read queue, **ptem** sends an **M_IOCACK** message downstream and an **M_CTL** message (with ioc_cmd set to **MC_DO_CANON** or **MC_NO_CANON**) upstream to enable or disable the input processing on **ldterm**.

- When an **M_DELAY** message is received on its read or write queue, **ptem** simply discards the message without any action.

- When an **M_IOCTL** message of type **JWINSIZE** is received on its write queue and if the values in the **jwinsize** structure in **ptem** are not zero, **ptem** sends an **M_IOCACK** message upstream with the **jwinsize** structure. If the values are zero, **ptem** sends an **M_IOCNAK** message upstream.

- When an **M_IOCTL** message of type **TIOCGWINSZ** is received on its write queue and if the values in the **winsize** structure in **ptem** are not zero, **ptem** sends an **M_IOCACK** message upstream with the **winsize** structure. If the values are zero, **ptem** sends an **M_IOCNAK** message upstream.
- When an **M_IOCTL** message of type **TIOCSWINSZ** is received in its write queue, **ptem** saves the information passed to it in the **winsize** structure and sends an **M_PCSIG** (with the signal number set to **SIGWINCH**) upstream to the pty slave process if the window size is changed.
- When an **M_IOCTL** message of type **TIOCGWINSZ** is received on its read queue and if the values in the **winsize** structure in **ptem** are not zero, **ptem** sends an **M_IOCACK** message downstream with the **winsize** structure. If the values are zero, **ptem** sends an **M_IOCNAK** message downstream.
- When an **M_IOCTL** message of type **TIOCSWINSZ** is received in its read queue, **ptem** saves the information passed to it in the **winsize** structure and sends an **M_PCSIG** (with the signal number is set to **SIGWINCH**) upstream to the pty slave process if the window size is changed.
- All other messages not mentioned above are passed to the next module or driver.

AUTHOR

ptem was developed by HP.

SEE ALSO

ioctl(2), streamio(7), ptm(7), pts(7), ldterm(7).

NAME

ptm - STREAMS master pty (pseudo-terminal) driver

SYNOPSIS

```
#include <sys/stropts.h>
#include <sys/ptyio.h>
#include <sys/strtio.h>

int open("/dev/ptmx", O_RDWR);
```

DESCRIPTION

A pseudo-terminal (pty) consists of a tightly-coupled pair of character devices, called the master device and slave device. The pty master and slave device drivers work together to simulate a terminal connection where the master provides a connection to the pseudo terminal server process and the slave provides a terminal device special file access for the terminal application processes, as depicted below:

```
                      ----------------
                     | pty functions  |
Application <-->     |----------------| <--> Server
 Processes           | Slave | Master |       Process
                     | (pts) | (ptm)  |
                      ----------------
```

The slave driver, **pts** with **ptem** (STREAMS pty emulation module) and **ldterm** (STREAMS line discipline module) pushed on top (not shown for simplicity), provides a terminal interface as described in *termio*(7). Whereas devices that provide the terminal interface described in *termio*(7) have a hardware device behind them; in contrast, the slave device has another process manipulating it through the master side of the pty. Data written on the master device is given to the slave device as input and data written on the slave device is presented as input on the master device.

In order to use the STREAMS pty subsystem, a node for the master pty driver **/dev/ptmx** and *N* number of slave pty devices must be installed (see *pts*(7) for details on slave pty). There are no nodes in the file system for each individual master device. Rather, the master driver is set up as a STREAMS clone driver (see *clone*(7)) with its major device number set to the major for the clone driver and its minor device number set to the major for the **ptm** driver. The master driver is opened using the **open()** system call with **/dev/ptmx** as the device file parameter. The clone open finds the next available minor number for the master device. The master device is available only if it and its corresponding slave device are not already opened. Only one open is allowed on a master device whereas multiple open are allowed on the slave device. When the master device is opened, the corresponding slave device is automatically locked out (see *pts*(7) on how to unlock the slave and obtain the slave device name). After both the master and slave have been opened, the user has two file descriptors which represent the end points of a full duplex connection composed of two streams. These two streams are automatically connected by the master and slave devices when they are opened. The user may then push the necessary modules on the master and slave streams (e.g., **ptem** and **ldterm,** on **pts** for terminal semantics, and **pckt** on **ptm** for Packet Mode feature).

The master and slave drivers pass all STREAMS messages to their adjacent drivers. Only the **M_FLUSH** message needs some special processing because the read queue of the master is connected to the write queue of the slave and vice versa. Hence, the **FLUSHR** flag is changed to **FLUSHW** flag and vice versa whenever a **M_FLUSH** message travels across the master–slave link. When the master device is closed, an **M_HANGUP** message is sent to the corresponding slave device which will render that slave device unusable. The process on the slave side gets the errno [ENXIO] when attempting a **write()** system call on the slave device but it will be able to read any data remaining on the slave stream. Finally, when all the data have been read, the **read()** system call will return 0 (zero) indicating that the slave can no longer be used. On the last close of the slave device, a zero-length **M_DATA** message is sent to the corresponding master device. When the application on the master side issues a **read()** or **getmsg()** system calls and a 0 is returned. The user of the master device decides whether to close the master device file which will dismantle the streams on the master side. If the master device remains opened, the corresponding slave device can be opened and used again by another user.

Unlike the slave device, the master device does not act like a terminal. If **O_NDELAY** or **O_NONBLOCK** is set, a read on the master device returns -1 with errno set to [EAGAIN] if no data is available, and a write returns –1 with errno set to [EAGAIN] if there is internal flow control on the stream.

The master **ptm** driver supports the following **ioctl()** requests:

ISPTM Determines whether the file descriptor is that of an open master device. On success, it returns the major and minor number (type dev_t) of the master device which can be used to determine the name of the corresponding slave device. On failure, it returns –1 with errno set to [EINVAL]. **ISPTM** on HP-UX can return valid device number with negative value. For example, with major number of the STREAMS pty master being 0x9c, **ICPTM** will return 0x9C000000 which is a negative number. Therefore, it is imperative that applications check for an explicit –1 instead of "< 0" (less than 0) on the return value.

ISPTM is used by functions **grantpt()**, **unlockpt()**, and **ptsname()**. User applications normally do not need to invoke this ioctl. The format of this ioctl is:

```
int ioctl(master_fd, ISPTM, 0)
```

UNLKPT Unlocks the master and the corresponding slave devices. On success, it returns 0. On failure, it returns –1 with errno set to [EINVAL]. **UNLKPT** is used by function **unlockpt()**. User applications normally do not need to invoke this ioctl. The format of this ioctl is:

```
int ioctl(master_fd, UNLKPT, 0)
```

TIOCREMOTE This ioctl puts the STREAMS pty in and out of Remote Mode. When Remote Mode is on, input data will be flow-controlled and passed through **ldterm** without any input processing regardless of the terminal mode. When the pty master driver receives this ioctl, it will send an **M_CTL** message downstream to **ldterm** via **ptm**, **pts**, and **ptem**. The command in the **M_CTL** message is set to **MC_NO_CANON** or **MC_DO_CANON** depending whether to turn on or off the Remote Mode. The format of this ioctl is:

```
int ioctl(master_fd, TIOCREMOTE, argument)
```

where the argument is set to 1 to turn on Remote Mode and 0 to turn it off. Remote Mode is normally used when doing remote line editing in a window manager, or whenever flow-controlled input is required. Each write to the master device produces a record boundary for the process reading the slave devices. In normal usage, a write of data is like the data typed as a line on the terminal; a write of 0 (zero) bytes is like typing an **EOF** (End-of-File) character.

TIOCSIGNAL This ioctl allows the master process to send a signal to the slave process. The format of this ioctl is:

```
int ioctl(master_fd, TIOCSIGNAL, argument)
```

where the argument is the signal number as defined in the header file **<sys/signal.h>**. For example the master process can send an **SIGINT** signal to the slave process by doing:

```
ioctl(master_fd, TIOCSIGNAL, SIGINT)
```

p

AUTHOR

ptm was developed by HP and OSF.

FILES

/dev/ptmx Streams pty master clone device

/dev/pts/*N* Streams pty slave devices (0 <= *N* < **NSTRPTY**), where **NSTRPTY** is a kernel tunable parameter which can be changed via SAM.

SEE ALSO

insf(1M), getmsg(2), ioctl(2), open(2), read(2), write(2), grantpt(3C), ptsname(3C), unlockpt(3C), clone(7), ldterm(7), pckt(7), ptem(7), pts(7), streamio(7), termio(7).

NAME

pts - STREAMS slave pty (pseudo-terminal) driver

SYNOPSIS

```
#include <sys/stropts.h>
#include <sys/termios.h>
#include <sys/strtio.h>

int open("/dev/pts/N", O_RDWR);
```

DESCRIPTION

A pseudo-terminal (pty) consists of a tightly-coupled pair of character devices, called the master device and slave device. The pty master and slave device drivers work together to simulate a terminal connection where the master provides a connection to the pseudo terminal server process and the slave provides a terminal device special file access for the terminal application processes, as depicted below:

```
                        ----------------
                       | pty functions  |
Application <-->       |----------------| <--> Server
 Processes             | Slave | Master |        Process
                       | (pts) | (ptm)  |
                        ----------------
```

The slave driver, **pts** with **ptem** (STREAMS pty emulation module) and **ldterm** (STREAMS line discipline module) pushed on top (not shown for simplicity), provides a terminal interface as described in *termio*(7). Whereas devices that provide the terminal interface described in *termio*(7) have a hardware device behind them; in contrast, the slave device has another process manipulating it through the master side of the pty. Data written on the master device is given to the slave device as input and data written on the slave device is presented as input on the master device.

In order to use the STREAMS pty subsystem, a node for the master pty driver **/dev/ptmx** and *N* number of slave pty devices must be installed (see *ptm*(7) for more details on master pty). When the master device is opened, the corresponding slave device is automatically locked out. No user can open that slave device until its permissions are changed (via the **grantpt()** function) and the device is unlocked (via the **unlockpt()** function). The user then call the **ptsname()** function to obtain the name of the slave device and invoke the **open()** system call to open the slave device. Although only one open is allowed on a master device, multiple opens are allowed on the slave device. After both the master and slave have been opened, the user has two file descriptors which represent the end points of a full duplex connection composed of two streams that are automatically connected by the master and slave devices when they are opened. The user may then push the desired modules (for example, **ptem** and **ldterm,** on **pts** for terminal semantics and **pckt** on **ptm** for Packet Mode feature).

The master and slave drivers pass all STREAMS messages to their adjacent drivers. Only the **M_FLUSH** message needs some special processing because the read queue of the master is connected to the write queue of the slave and vice versa. For example, the **FLUSHR** flag is changed to **FLUSHW** flag and vice versa whenever a **M_FLUSH** message travels across the master–slave link. When the master device is closed, an **M_HANGUP** message is sent to the corresponding slave device which will render that slave device unusable. The process on the slave side gets the errno [ENXIO] when attempting a **write()** system call to the slave device file but it will be able to read any data remaining in the slave stream. Finally, when all the data has been read, the **read()** system call will return 0, indicating that the slave can no longer be used. On the last close of the slave device, a zero-length **M_DATA** message is sent to the corresponding master device. When the application on the master side issues a *read*(2) or *getmsg*(2) system calls, a 0 (zero) is returned. The user of the master device may decide to close the master device file, which dismantles the stream on the master side. If the master device remains opened, the corresponding slave device can be opened and used again by another user.

EXAMPLES

The following example shows how a STREAMS pty master and slave devices are typically opened.

```
int fd_master, fd_slave;
char *slave;
   ...
     fd_master = open("/dev/ptmx", O_RDWR);
     grantpt(fd_master);
     unlockpt(fd_master);
```

p

```
slave = ptsname(fd_master);
fd_slave = open(slave, O_RDWR);
ioctl(fd_slave, I_PUSH, "ptem");
ioctl(fd_slave, I_PUSH, "ldterm");
```

AUTHOR

pts was developed by HP and OSF.

FILES

/dev/ptmx Streams pty master clone device

/dev/pts/*N* Streams pty slave devices (0 <= *N* < **NSTRPTY**), where **NSTRPTY** is a kernel tunable parameter which can be changed via SAM (see *sam*(1M)).

SEE ALSO

insf(1M), sam(1M), getmsg(2), ioctl(2), open(2), read(2), write(2), grantpt(3C), ptsname(3C), unlockpt(3C), ldterm(7), ptem(7), ptm(7), streamio(7), termio(7).

p

NAME

pty - pseudo-terminal driver

DESCRIPTION

The **pty** driver provides support for a device-pair termed a pseudo terminal. A pseudo terminal is a pair of character devices, a master device and a slave device. The slave device provides to application processes an interface identical to that described in *termio*(7). Unlike all other devices that provide the interface described in *termio*(7), the slave device does not have a hardware device behind it. Instead, it has another process manipulating it through the master half of the pseudo terminal. Thus anything written on the master device is given to the slave device as input, and anything written on the slave device is presented as input on the master device.

```
                       ----------------
                      | pty functions  |
  Application <-->    |----------------| <--> Server
   Processes          | Slave | Master |        Process
                      | (pts) | (ptm)  |
                       ----------------
```

Open and Close Processing

The slave side of the **pty** interprets opening or closing the master side as a modem connection or disconnection on a real terminal. Only one open to the master side of a **pty** is permitted. An attempt to open an already open master side returns **-1** and sets the external variable **errno** to [EBUSY]. An attempt to open the master side of a **pty** that has a slave with an open file descriptor returns **-1** and sets **errno** to [EBUSY]. The potential problem of **pty**s being found busy at opens can be avoided by using the *clone open* functionality discussed in the next section.

An attempt to open a nonexistent **pty** returns **-1** and sets **errno** to [ENXIO]. If **O_NDELAY** is not specified, opens on the slave side hang until the master side is opened. If **O_NDELAY** is specified, opens on the slave side return error if the master side is closed. Any **ioctl()** or **write()** request made on the slave side of a **pty** after the master side is closed returns **-1** and sets the external variable **errno** to [EIO]. A **read()** request made on the slave side of a **pty** after the master side is closed returns 0 bytes. Closing the master side of a **pty** sends a **SIGHUP** hangup signal to the tty process group number of the corresponding slave side and flushes pending input and output.

Clone Open

In typical **pty** usage, there is no preference among **pty** pairs. Thus, it is useful to be able to issue a single **open()** that internally opens any available **pty**. An open on **/dev/ptym/clone** returns an open file descriptor of a free master **pty** device. If there are no free devices, the open returns **-1** and sets **errno** to [EBUSY]. The name of the slave device corresponding to the opened master device can be found through a **ptsname()** request.

Processing ioctl() Requests

By default, any **ioctl()** request defined by *termio*(7) is recognized by both the master and slave sides of a **pty**. These **ioctl()** requests are processed by the **pty** driver as specified by *termio*(7). In addition, the **ioctl()** requests defined below are recognized by the master side of a **pty**. The slave side only recognizes **ioctl()** requests defined by *termio*(7). An **ioctl()** request made on the slave side of a **pty** after the master side is closed returns **-1** and sets the external variable **errno** to [EIO]. An **ioctl()** request not recognized by the **pty** returns **-1** and sets the external variable **errno** to [EINVAL]. Note that some of the master-side-only **ioctl()** requests affect which **ioctl()** requests are recognized by the master and slave side of the **pty**. These master-side-only **ioctl()** requests also affect the way recognized **ioctl()** requests, **open()** requests, and **close()** requests are processed by the **pty** driver.

The following **ioctl()** requests, defined in **<sys/ptyio.h>**, apply only to the master side of **pty**:

TIOCSIGSEND
: Cause a signal to be sent from the slave side of the **pty** to the current tty process group of the slave side. The value of the parameter is taken to be the signal number sent. An [EINVAL] error is returned and no signal is sent if the specified signal number does not refer to a legitimate signal (see *signal*(5)). Note that this request allows the server process to send signals to processes not owned by the same user ID.

TIOCTTY
: Enable or disable all **termio** processing by a **pty**. **termio** processing is enabled if the **int** addressed by *arg* is nonzero and disabled if the **int** addressed by *arg* is zero. By

default, **termio** processing is enabled. **termio** processing refers to processing of input and output described by *termio*(7) (such as tab expansion), as well as the processing of the **ioctl()** requests described by *termio*(7). When disabled, all input and output data is passed through the **pty** without modification. Issuing a **TIOCTTY ioctl()** request flushes all data buffered in the pseudo terminal and releases any processes blocked waiting for data. Enabling and disabling **TIOCTTY** affects the operation of the following **ioctl()** requests: **TIOCPKT**, **TIOCREMOTE**, **TIOCBREAK**, **TIOCSTOP**, **TIOCSTART**, **TIOCTRAP**, and **TIOCMONITOR**.

When **TIOCTTY** is enabled, all **termio ioctl()** requests execute as specified in *termio*(7), regardless of the side from which the **ioctl()** request is made. When **TIOCTTY** is disabled, master side **termio ioctl()** requests set and return the the external variable **errno** to [EINVAL]. Slave side **termio ioctl()** requests are processed like any other **ioctl()** request when **TIOCTTY** is disabled. In particular, slave side **termio ioctl()** requests set and return the external variable **errno** to [EINVAL] when both **TIOCTTY** and **TIOCTRAP** are disabled. (See the discussion of **ioctl()**, **open()**, and **close()** trapping below). **ioctl()** requests not defined by *termio*(7) are not affected by the state of **TIOCTTY**.

Data written through a pseudo terminal with **TIOCTTY** disabled is handled in a manner similar to data flowing through a pipe. A write request blocks in the **pty** until all data has been written into the **pty**. A read request blocks if there is no data available unless the **O_NDELAY** flag is set (see *fcntl*(2)). When data is available to be read, the read request returns whatever is available, and does not wait for the number of bytes requested to be satisfied. The number of bytes a **pty** can contain in its internal memory is implementation dependent, but is at least 256 bytes in each direction. For example, a write on the slave side of a **pty** of 1024 bytes might be read on the master side by four read requests returning 256 bytes each. The size of the chunks of data that are read is not guaranteed to be consistent, but no data is lost.

The following **ioctl()** requests, defined in **<sys/ptyio.h>**, apply only to the master side of a **pty**. In particular, these **ioctl()** requests enable/disable specific modes of **pty** driver operation. These **ioctl()** requests work in series with **TIOCTTY**; that is, the mode must be enabled by its **ioctl()** request and **TIOCTTY** must be enabled for the mode to operate. The mode can be enabled or disabled regardless of the state of **TIOCTTY**.

p

TIOCPKT Enable or disable packet mode. Packet mode is enabled if the **int** addressed by *arg* is nonzero and disabled if the **int** addressed by *arg* is zero. By default, packet mode is disabled. When applied to the master side of a pseudo terminal, each subsequent **read()** from the master side returns data written on the slave part of the pseudo terminal preceded by a zero byte (symbolically defined as **TIOCPKT_DATA**), or a single byte reflecting control status information. The value of such a status byte is composed of zero or more bit flags:

TIOCPKT_FLUSHREAD
The read queue for the slave side has been flushed.

TIOCPKT_FLUSHWRITE
The write queue for the slave side has been flushed.

TIOCPKT_STOP
Data flowing from the slave side of the **pty** to the master side has been stopped by means of **^S**, **TIOCSTOP**, or **TCXONC**.

TIOCPKT_START
Data flowing from the slave side of the **pty** to the master side has been restarted.

TIOCPKT_DOSTOP
Stop and start characters have been set to **^S** or **^Q**.

TIOCPKT_NOSTOP
Stop and start characters are set to something other than **^S** or **^Q**.

TIOCREMOTE Enable or disable remote mode. Remote mode is enabled if the **int** value of *arg* is nonzero and disabled if the **int** value of *arg* is zero. By default, remote mode is disabled. Remote mode is independent of packet mode. This mode causes input to the pseudo terminal to be flow controlled and not input edited (regardless of the terminal mode). Each write to the master side produces a record boundary for the process reading the slave side. In normal usage, writing data is like typing the data as a line on a terminal; writing zero bytes is equivalent to typing an end-of-file character (that is, the EOF character as defined in *termio*(7)). The data read by the slave side is identical to the data written on the master side. Data written on the slave side and read on the master side with **TIOCREMOTE** enabled is still subject to the normal *termio*(7) processing. **TIOCREMOTE** can be used when doing remote line editing in a window manager, or whenever flow-controlled input is required. Issuing a **TIOCMONITOR ioctl()** request flushes all data buffered in the pseudo terminal.

The following **ioctl()** requests, defined in **<sys/ptyio.h>**, apply only to the master side of **pty**. In particular, these **ioctl()** requests are only recognized when **TIOCTTY** is enabled. When **TIOCTTY** is disabled, these **ioctl()** requests set and return the external variable **errno** to [EINVAL].

TIOCBREAK Cause a break operation to be done on the slave side of the **pty**, as if a user had pressed the break key on a real terminal. Takes no parameter.

TIOCSTOP Stop data flowing from the slave side of the **pty** to the master side (equivalent to typing **^S**). Takes no parameter.

TIOCSTART Restart output (stopped by **TIOCSTOP** or by typing **^S**). Takes no parameter.

Flow-Control Input and Output Processing

The following terms are used to describe the flow of data through pseudo terminals. INPUT refers to data flowing from the master side of a **pty** to the slave side. OUTPUT refers to data flowing from the slave side of a **pty** to the master side.

When packet mode (**TIOCPKT**) is disabled and INPUT is stopped (see IXOFF, input modes, in *termio*(7)), the next **read()** from the master side of a **pty** returns a STOP character. When INPUT is restarted, the next **read()** from the master side returns a START character. If packet mode (**TIOCPKT**) is enabled, the STOP or START character is preceded by a data packet indicator (**TIOCPKTDATA**). **select()** should be used by the master-side server before each **write()** request to properly handle INPUT flow control (see *select*(2)).

When INPUT flow control is enabled, **write()** and **select()** are handled as follows: Write-selects on the master side of a **pty** return true only if INPUT has not been stopped. If INPUT becomes stopped while data is being written into the master side of a **pty**, the write returns with the number of bytes written before INPUT was stopped. Writes done after INPUT is stopped return immediately with zero bytes written.

When packet mode (**TIOCPKT**) is disabled and OUTPUT is stopped (see IXON, input modes in *termio*(7)), each subsequent **read()** from the master side of a **pty** returns with no data read. When OUTPUT is restarted, each subsequent **read()** from the master side returns data written on the slave side. If packet mode (**TIOCPKT**) is enabled, the first **read()** after OUTPUT has been stopped returns a **TIOCPKTSTOP** packet. All subsequent reads from the master side while OUTPUT is stopped returns a **TIOCPKTDATA** packet with no data. When OUTPUT is restarted, the next **read()** from the master side returns a **TIOCPKTSTART** packet. All subsequent reads from the master side return data written on the slave side preceded by a **TIOCPKTDATA** packet. **select()** should be used by the master-side server before each **read()** to properly handle OUTPUT flow control. Otherwise, reads from the master side of a **pty** will not be prevented when OUTPUT is stopped.

Trapping ioctl(), open(), close() Requests

When trapping is enabled, the master side is notified when the application on its slave side makes an **ioctl()**, **open()**, or **close()** request. For trapped **ioctl()** and **open()** requests, the slave side is blocked (that is, the request does not complete) until the server on its master side acknowledges the trapped request. For trapped **close()** requests, the slave slave does not block for an acknowledgement.

select() should be used by the master side server to receive notification of trapped **ioctl()**, **open()**, and **close()** requests. When one of these requests is trapped, the **select()** returns with an "exceptional condition" indicated for the slave side's file descriptor. Other mechanisms for receiving notification of trapped requests are defined below, but these mechanisms should be used only if **select()** is not available.

When trapping is disabled (default condition), unrecognized slave **ioctl()** requests return an error, with the external variable **errno** set to [EINVAL]. The only **ioctl()** requests recognized by the slave side are those defined by *termio*(7) and only when **TIOCTTY** is enabled. When **TIOCTTY** is disabled, no **ioctl()** requests are recognized by the slave side. If trapping is enabled and the master side closes, trapping is disabled. If the master closes during the middle of a handshake with the slave, the handshake

p

is done automatically.

Trapping occurs in two forms that are identified by the **ioctl()** requests that enable or disable them — **TIOCTRAP** and **TIOCMONITOR**. These two forms are distinguished by the types of requests they affect and by the capabilities they provide. Trapping **open()** and **close()** requests is enabled or disabled by **TIOCTRAP**. Trapping **ioctl()** requests not defined by *termio*(7) are enabled or disabled by **TIOCTRAP**. Trapping **ioctl()** requests defined by *termio*(7) are enabled or disabled by **TIOCTRAP** only when **TIOCTTY** is also disabled. When **TIOCTTY** is enabled, trapping **ioctl()** requests defined by *termio*(7) are enabled or disabled by **TIOCMONITOR**. Briefly, both **TIOCTRAP** and **TIOCMONITOR** trapping allow the server on the master side to examine the request's parameters, the pid making the request, etc. In addition, **TIOCTRAP** trapping allows the server to modify the parameters and return values of an **ioctl()** request.

The following **ioctl()** calls apply only to the master side of a **pty** and pertain to trapping **ioctl()**, **open()**, and **close()** requests. They are defined in **<sys/ptyio.h>**:

TIOCTRAP Enable or disable trapping of **ioctl()**, **open()**, and **close()** requests made by the application on the slave side of a **pty**. Trapping is enabled if the **int** addressed by *arg* is nonzero and disabled if the **int** addressed by *arg* is zero. By default, **TIOCTRAP** trapping is disabled.

TIOCTRAPSTATUS
Check for a pending **ioctl()**, **open()**, or **close()** trap. The argument points to an **int** that is set to one if a trap is pending and to zero if nothing is pending. Use **TIOCTRAPSTATUS** when the preferred method of a **select()** "exceptional condition" is not available.

TIOCREQCHECK
Return the trapped **ioctl()**, **open()**, or **close()** information to the master side. Use **TIOCREQCHECK** in response to either a **select()** "exceptional condition" or a **TIOCTRAPSTATUS** indicating that a trap is pending. A **TIOCREQCHECK** reads the pending **ioctl()**, **open()**, or **close()** information into the memory pointed to by the *arg* of **TIOCREQCHECK**. The information takes the form of the following **request_info** structure, defined in **<sys/ptyio.h>**:

```
struct request_info {
    int request;
    int argget;
    int argset;
    pid_t pgrp;
    pid_t pid;
    int errno_error;
    int return_value;
};
```

All elements of **request_info** refer to the slave side of the **pty** and include the following:

request The **ioctl()** command received.

argget The **ioctl()** request applied to master side to receive the trapped **ioctl()** structure, if one exists (a zero value means there is none). (When nonzero, **argget** is a **TIOCARGGET** request with the size field precomputed.)

argset The **ioctl()** request applied to master side to send back the resulting **ioctl()** structure, if one exists (a zero value means there is none). (When nonzero, **argset** is a **TIOCARGSET** request with the size field precomputed.)

pgrp The process group number of the process doing the operation.

pid The process ID of the process doing the operation.

errno_error
The **errno** external variable error code (initialized to zero) returned by **ioctl()** on the slave side. When open error mode is enabled,

p

errno_error can be used to return an error for trapped slave **pty open()** requests. See the discussion of the **TIOCSMODES ioctl()** for further information on open error mode.

return_value
The success value (initialized to zero) returned by **ioctl()** on the slave side when **errno_error** is not set.

When the **ioctl()** argument received on the slave side is not a pointer, its value is stored as four bytes retrievable with an **ioctl()** request to the master side equal to **argget**.

When an **open()** or **close()** is being passed, **request** is set to **TIOCOPEN** or **TIOCCLOSE**, respectively. For **TIOCOPEN** and **TIOCCLOSE**, both **argget** and **argset** are zero because there is no **ioctl()** structure. When **TIOCTTY** is enabled, the *termio*(7) definition of open/close is executed first before being passed to the master side. Note that while all opens are trapped, only the last close on a particular inode for a **pty** slave side is trapped by the **pty**.

A **TIOCREQCHECK** returns the external variable **errno** error [EINVAL] if no **ioctl()**, **open()**, or **close()** trap is pending. Accordingly, a **TIOCREQCHECK** that returns [EINVAL] in response to a **select()** "exceptional condition" indicates that the trapped **ioctl()**, **open()**, or **close()** request was terminated by a signal after **select()** returned.

TIOCREQGET Identical to **TIOCREQCHECK** except when no **ioctl()**, **open()**, or **close()** trap is pending. A **TIOCREQGET** blocks until a slave side **ioctl()**, **open()**, or **close()** is trapped; whereas a **TIOCREQCHECK** returns [EINVAL]. Use **TIOCREQGET** when neither the preferred method of a **select()** "exceptional condition" nor the master side **ioctl() TIOCTRAPSTATUS** is available.

TIOCREQSET Complete the handshake started by a previous **TIOCREQCHECK** or **TIOCREQGET**. The argument should point to the **request_info** structure, as defined by the **TIOCREQCHECK**.

Before doing this **ioctl()** request to complete the handshake, the server should set **errno_error** to an external variable **errno** error value to be passed back to the slave side. If there is no error, **errno_error** can be left alone because the **pty** initializes it to zero. Also, when there is no error, **return_value** should be set if other than a zero result is desired. The server can set **return_value** and **errno_error** if the trapped request is an **ioctl()** and may set **errno_error** for a trapped **open()** if open error mode is enabled. Setting either **return_value** or **errno_error** for a trapped **close()** affects neither the return value of the request nor the external variable **errno** value of the slave side. Setting either **return_value** or **errno_error** for a trapped **open()** affects neither the return value of the request nor the external variable **errno** value of the slave side unless open error mode is enabled. Open error mode allows the server to return an error to a trapped slave **open()** by setting **errno_error**. Unlike **ioctl()** requests, setting **return_value** never affects slave **pty open()** requests. Further, setting either **return_value** or **errno_error** does not cause **TIOCREQSET** to return an error to the server.

If the **TIOCREQSET** request is made and the request value in the passed **request_info** structure does not equal the trapped value, the external variable **errno** is set and returned as [EINVAL]. [EINVAL] is also returned if there are no trapped **ioctl()**, **open()**, or **close()** requests. If the trapped request has been interrupted by a signal between the time that the server has done the **TIOCREQGET** and the **TIOCREQSET**, the **TIOCREQSET** request returns [EINVAL].

TIOCGFLAGS Get the file status flags associated with a trapped request. Upon successful return, the **ioctl()** returns in an integer referenced by *arg* the file status flags for the trapped request. The flag definitions in **<sys/file.h>** can be used to interpret the flags. If no trap is currently pending, the **TIOCGFLAGS ioctl()** returns an error with the external variable **errno** set to [EINVAL].

TIOCMONITOR
Enable or disable read-only trapping of **termio ioctl()** requests. **TIOCMONITOR**

p

trapping is enabled if the **int** addressed by *arg* is nonzero and disabled if the **int** addressed by *arg* is zero. By default, **TIOCMONITOR** trapping is disabled. **TIOCMONITOR** works in series with **TIOCTTY**; that is, the **TIOCMONITOR** trapping must be enabled and **TIOCTTY** must be enabled for **termio ioctl()** requests to be trapped by **TIOCMONITOR**. **TIOCMONITOR** trapping can be enabled or disabled regardless of the state of **TIOCTTY**.

When **TIOCTTY** is disabled, **termio ioctl()** requests are not trapped by **TIOCMONITOR**. However, **ioctl()** requests are trapped by **TIOCTRAP** if **TIOCTTY** is disabled and **TIOCTRAP** is enabled. **TIOCTRAP** trapping allows the master side server to modify the parameters and return values of an **ioctl()** request, whereas **TIOCTMONITOR** trapping does not.

TIOCMONITOR trapping allows the server on the master side to know when characteristics of the line discipline in the **pty** are changed by an application on its slave side. The mechanism for handshaking **termio** requests trapped by **TIOCMONITOR** is the same as the mechanism described above for requests trapped by **TIOCTRAP**. (It is recommended that **termio ioctl()** requests be used on the master side to interrogate the configured state of the line discipline in the **pty**. This compensates for the window of time before **TIOCMONITOR** is enabled, when **termio ioctl()** requests are not trapped.)

When using **select()** on the master side of a **pty**, the "exceptional condition" refers to an **open()**, **close()**, or **ioctl()** request pending on the slave side, while "ready for reading or writing" indicates that the device can be read from or written to successfully.

Of the **ioctl()** requests subject to being trapped, only one-per-pty can be handled at a time. This means that when an application does a non-**termio ioctl()** request to the slave side, all other **ioctl()** requests to the same **pty** slave side are blocked until the first one is handshaked back by the master side. (**ioctl()** requests that are not trapped, such as **termio** when **TIOCTTY** is enabled and **TIOCMONITOR** is disabled, are not blocked.) This permits the implementation of indivisible operations by an **ioctl()** call on the slave side that is passed to the server process.

In summary, the following method of handling trapped **ioctl()**, **open()**, and **close()** requests is preferred:

1. Call **select()**. This system call blocks the master side until a slave side **ioctl()**, **open()**, or **close()** request is trapped.
2. Make **TIOCREQCHECK ioctl()** request. This step returns information about a trapped **ioctl()**, **open()**, or **close()** request. If **TIOCREQCHECK** returns the external variable **errno** error [EINVAL], loop back to the **select()** call.
3. Make **argget ioctl()** request. This optional step is used if **argget** is nonzero and the server wants to do more than just reject the trapped slave **ioctl()** request.
4. Make **argset ioctl()** request. This optional step is done if **argset** is nonzero and the server wants to pass back a modified **ioctl()** structure. It is done after the trapped **ioctl()** request is processed via the server on the master side.
5. Set **errno_error** and **return_value**. If the trapped request is an **ioctl()**, set **errno_error** appropriately. If the appropriate value for **errno_error** is zero, **return_value** must be set. If open error mode is enabled, set **errno_error** to a nonzero value to return an error to a trapped **open()** request.
6. Make **TIOCREQSET ioctl()** request. This step completes the trapped **ioctl()**, **open()**, or **close()** request.

While a process is waiting in the slave side of the **pty** for the server to complete a handshake, it is susceptible to receiving signals. The following master side **ioctl()** request allows the server process to control how the **pty** responds when a signal attempts to interrupt a trapped **open()** or **ioctl()** request:

TIOCSIGMODE

Set the signal handling state of the **pty** to the mode specified as the argument. The mode can have three values, which are **TIOCSIGBLOCK**, **TIOCSIGABORT**, and **TIOCSIGNORMAL**.

TIOCSIGBLOCK

Cause some signals to be postponed that are destined for the slave-side process whose **open()** or **ioctl()** request is trapped. Signals are postponed if they would

p

otherwise cause the process to jump to an installed signal handler. Signals are not postponed if they would otherwise cause the process to abort or if they are being ignored. When the server process completes the handshake by means of the **TIOCREQSET ioctl()** request, the process returns to the calling program and any pending signals are then acted upon. Any signals that the user has blocked by means of **sigblock()** continues to be blocked.

TIOCSIGABORT
Prevent a trapped **open()** or **ioctl()** request from being restarted. The server process sets this mode when it wants the interrupted requests to return to the calling program with an [EINTR] error.

TIOCSIGNORMAL
This is the default mode of the **pty**. If a signal interrupts a trapped **open()** or **ioctl()** request, the user's signal handler routine can specify whether the request is to be restarted. If the request is restarted, it executes again from the beginning and the server has to make another **TIOCREQGET** request to start the handshake over again. If the user's signal handler routine specifies that the interrupted request should not be restarted, the request returns to the calling program with [EINTR] upon completion of the signal handler. Note that the restarted request is not necessarily the very next one to be trapped.

The following **ioctl()** requests, defined in **<sys/ptyio.h>**, provide a mechanism to get and set **pty** modes. Five of the modes can also be manipulated using other **ioctl()** requests discussed previously. See the bit definitions for the **ioctl()** equivalents. The effect of enabling or disabling them by either means is identical. Commonly, an application would use the **TIOCGMODES ioctl()** to get the **pty** modes currently in effect, set or clear the bits for the modes being changed, and issue a **TIOCGMODES ioctl()** to effect the desired change.

TIOCGMODES Get the **pty** modes currently in effect. The **ioctl()** returns in a long referenced by *arg* bits indicating the states of various **pty** modes. If a bit is set, the associated mode is enabled. If a bit is clear, the associated mode is disabled. Unused bits are clear. The meaning of the bits is described under the description of the **TIOCSMODES ioctl()**.

TIOCSMODES Set the **pty** modes according to the value of type long referenced by *arg*. Unused bits are ignored but should be set to zero. The bit values for **pty** modes are listed below.

PM_REMOTE
Enable or disable remote mode. See the discussion of the **TIOCREMOTE ioctl()**.

PM_TTY
Enable or disable tty mode. See the discussion of the **TIOCTTY ioctl()**.

PM_PKT
Enable or disable packet mode. See the discussion of the **TIOCPKT ioctl()**.

PM_TRAP
Enable or disable trap mode. See the discussion of the **TIOCTRAP ioctl()**.

PM_MONITOR
Enable or disable monitor mode. See the discussion of the **TIOCMONITOR ioctl()**.

PM_OPEN_ERROR
Enable or disable open error mode. Open error mode allows a server process to return an error to a trapped slave **pty open()** through the **TIOCREQSET ioctl()**. When open error mode is enabled, the server may return a trapped **open()** with an error by setting the **errno_error** field in the **request_info** structure passed to the TIOCREQSET **ioctl()**. When open error mode is disabled (the default state), setting **errno_error** to handshake a slave **open()** has no effect. Note that unlike the **ioctl()** trap handshaking, setting **return_value** has no effect for a slave **open()** regardless of the state of open error mode. See the discussion of the **TIOCREQSET ioctl()** for further details on handshaking a trapped request.

WARNINGS

The slave side cannot indicate an end-of-file condition to the master side.

When using **TIOCREMOTE**, a single **write()** request to the master side of greater than 256 bytes may result in multiple smaller records being read from the slave side instead of only one record.

AUTHOR

pty was developed by the University of California, Berkeley.

FILES

/dev/ptym/pty[**a-ce-su-z**][**0-9**][**0-9**]	master pseudo terminals
/dev/ptym/pty[**a-ce-su-z**][**0-9**][**0-9**][**0-9**]	master pseudo terminals
/dev/ptym/pty[**a-ce-su-z**][**0-9a-f**]	master pseudo terminals
/dev/pty[**pqr**][**0-9a-f**]	master pseudo terminals
/dev/pty/tty[**a-ce-su-z**][**0-9**][**0-9**]	slave pseudo terminals
/dev/pty/tty[**a-ce-su-z**][**0-9**][**0-9**][**0-9**]	slave pseudo terminals
/dev/pty/tty[**a-ce-su-z**][**0-9a-f**]	slave pseudo terminals
/dev/tty[**pqr**][**0-9a-f**]	slave pseudo terminals

SEE ALSO

close(2), fcntl(2), ioctl(2), open(2), read(2), select(2), sigblock(2), write(2), ptsname(3C), signal(5), termio(7).

NAME

random, urandom, rng - strong random number generator

SYNOPSIS

#include <sys/random.h>

DESCRIPTION

The character special files **/dev/random** and **/dev/urandom** provide an interface to the kernel-resident random number generator, **rng**. A **read()** from **/dev/random** is potentially blocking. A **read()** from **/dev/urandom** is always nonblocking. Data from **/dev/urandom** can potentially have lower entropy than data from **/dev/random**.

The **rng** module is a dynamically loadable kernel module (DLKM). That is, it can be dynamically unconfigured or reconfigured by an administrator with root authority without rebooting the system.

A sequence from **rng** has unlimited entropy. In contrast, a sequence generated computationally by a pseudorandom number generator, such as *random*(3M), has limited entropy, derived only from its initial seed. The **rng** module should be considered a quality source for randomness. It has passed extensive statistical testing, including the NIST (National Institute of Standards and Technology) tests for randomness.

The **rng** module uses the uncertainty in completion times of interrupt threads triggered by external events. The **rng** module extracts a sequence of bits from the interrupt time stamps. Any existing bit bias is removed to yield a sequence with uniform distribution of 0's and 1's. The resulting sequence is divided between the holding buffers for the special files **/dev/random** and **/dev/urandom**. For each **read()** on **/dev/random** and **/dev/urandom**, data is retrieved from the corresponding holding buffer. A hash function based on AES (Advanced Encryption Standard) is applied and the result is placed in the buffer provided by the user. All requests on the holding buffers are serialized to ensure that returned random data is not shared between different requests even for simultaneous requests on a multiprocessor system.

There is no **write()** function associated with either **/dev/random** or **/dev/urandom**, and both devices are read-only by all users. A single **ioctl()** is defined for **/dev/random** to facilitate independent verification of **rng** production.

The file **/usr/include/sys/random.h** contains the following definitions:

```
/* The maximum request size, for read() or ioctl(), in bytes   */
#define RNG_READMAX    256

/* ioctl() to retrieve data from the entropy collector directly*/
#define RNG_GETRAW     _IOR('Q', 0, uint8_t[RNG_READMAX])
```

If a **read()** request is for more than RNG_READMAX bytes, it is treated as if it was for exactly **RNG_READMAX** bytes. This holds for both **/dev/random** and **/dev/urandom**.

Specific Information About /dev/random

When there are a large number of requests on **/dev/random** within a short time interval, the demand on the holding buffer can exceed the rate at which data is supplied by **rng**. A **read()** on the **/dev/random** device blocks the requesting thread if the random data stored in the holding buffer is too low to complete the request. The thread blocks until the holding buffer has been updated with enough random data to complete the request.

For **/dev/random open()** flags, only **O_NONBLOCK** and **O_NDELAY** have device-specific actions. If neither of these flags is set, a **read()** on **/dev/random** will block until the amount of data requested, up to **RNG_READMAX** bytes, can be returned. When the requested number of bytes is not available and either of the above flags are set, **read()** returns immediately. If the **O_NONBLOCK** flag is set, **read()** returns -1 and errno is set to **EAGAIN**. If **O_NONBLOCK** is not set and **O_NDELAY** is set, **read()** returns zero.

The **RNG_GETRAW ioctl()** permits an application with superuser privilege to fetch **RNG_READMAX** bytes of data directly from the **/dev/random** holding buffer, after bias has been removed but before the AES hash. This interface is not intended to be used for cryptographic applications, rather, for statistical testing of the randomness of the data in the **/dev/random** holding buffer. This **RNG_GETRAW ioctl()** blocks for the same reason as a read on **/dev/random**. If the requesting thread does not have superuser authority, **EACCES** is returned.

Specific Information About /dev/urandom

To address the limited random data collection rate problem, the **/dev/urandom** device is strictly non-blocking. The **/dev/urandom** holding buffer is regularly updated with random data, yet a high number of reads can decrease the entropy in its holding buffer. Under this conditions, the entropy of the data from **/dev/urandom** will be slightly lower that the one from **/dev/random**, yet **/dev/urandom** can still be considered a good source of random numbers.

There are no **open()** flags that result in device-specific actions with **/dev/urandom read()**.

ERRORS

[EAGAIN] For **/dev/random read()**, **O_NONBLOCK** was set when **/dev/random** was opened, and there is insufficient content in the holding buffer to complete the request.

[EACCES] For the **/dev/random RNG_GETRAW ioctl()**, the requesting thread did not have superuser authority.

AUTHOR

The random number generator was developed by HP.

For bias removal, the generator uses an algorithm by Dr. Yuval Perez, University of California.

The secure hashing uses an AES implementation provided by Dr. Brian Gladman, UK.

The NIST statistical tests are available at **http://csrc.nist.gov/rng**.

FILES

/dev/random

/dev/urandom

SEE ALSO

random(3M).

NAME

route - kernel packet forwarding database

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <net/route.h>
#include <net/if.h>

s = socket(AF_ROUTE, SOCK_RAW, family);
```

DESCRIPTION

This manpage describes routing socket interface to read and write kernel routing messages.

The information on how to transmit network packets is maintained by the HP-UX kernel in the routing information database, also known as the routing table. A user process can read or update information in the routing table by sending routing messages to the kernel via an AF_ROUTE socket. The message types are described in more detail in the *Message Types* section below.

The *family* parameter in the **socket** system call shown in the *SYNOPSIS* may be used to filter the routing messages the caller receives. The valid values for *family* are:

AF_INET	get routing messages affecting the Internet Protocol.
AF_INET6	get routing messages affecting the Internet Protocol version 6.
AF_UNSPEC	get routing messages affecting both AF_INET and AF_INET6 protocols.

Entries in the routing table specify the appropriate remote host or gateway to use when transmitting packets. These entries are either host-specific, or are applicable to all hosts located on a generic subnetwork, as specified by a netmask value.

After the system boots, each protocol family adds entries to the routing table for each network interface configured and ready to transmit network traffic. Normally, the route entry is specified as a **direct connection** to the destination host or network. For direct routes, the transport layer of the network stack sends packets directly to the host specified in the packet header. For non-direct routes, the interface forwards the packet to the gateway listed in the routing entry for that interface.

When routing packets, the kernel attempts to find an optimal route for each destination. If more than one entry matches the netmask of the destination, the kernel selects the route with the greater number of 1's in the netmask.

A default (wildcard) route is used if no other route to a particular remote host or network can be located. A default route is specified with an all 0 destination address value and a netmask of all 0's. Default routes, in combination with routing redirects, provide an economical mechanism for routing network traffic.

If no routing entry is found, the destination is declared as unreachable, and a routing-miss message (RTM_MISS) is generated to any user processes using the routing socket facilities, as described below.

Message Types

After creating a routing socket, the process can send commands to the kernel by writing to the socket. The process can read information from the kernel by reading from the socket. The following message types can be used to communicate routing information between the user process and the kernel:

RTM_ADD	add route
RTM_CHANGE	change gateway, metrics or flags
RTM_DELADDR	address being removed from interface
RTM_DELETE	delete route
RTM_GET	report metrics and other information
RTM_IFINFO	interface going up, down, etc.
RTM_LOSING	kernel suspects route is failing
RTM_LOCK	lock specified metrics
RTM_MISS	lookup on this address failed
RTM_NEWADDR	address being added to interface
RTM_REDIRECT	kernel instructs to use different route
RTM_RESOLVE	request to resolve destination to link-layer address

All 12 message types can be used to read information from the kernel. To write to the kernel, the process can issue RTM_ADD, RTM_DELETE, or RTM_GET message types to update information in the routing table.

Message types RTM_CHANGE and RTM_LOCK are not supported on HP-UX. If a user process issues these messages, [EOPNOTSUPP] error will be returned.

Message Structure

Messages are formed by a message header followed by a small number of socket address structures.

What message header to use depends on the message type. The RTM_IFINFO messages use the **if_msghdr** header. The RTM_NEWADDR and RTM_DELADDR messages use the **ifa_msghdr** header. All other message types use the **rt_msghdr** header.

The **rt_msghdr** structure contains the following members:

```
uint16_t rtm_msglen;       /* to skip over unrecognized messages */
uint8_t  rtm_version;      /* future binary compatibility */
uint8_t  rtm_type;         /* message type */
uint16_t rtm_index;        /* index for associated ifp */
int32_t  rtm_flags;        /* flags, incl. kern & message,
                            * e.g. DONE */
int32_t  rtm_addrs;        /* bitmask identifying sockaddrs in
                            * the message */
pid_t    rtm_pid;          /* identify sender */
int32_t  rtm_seq;          /* for sender to identify action */
int32_t  rtm_errno;        /* error indicator */
int32_t  rtm_use;          /* from rtentry */
uint32_t rtm_inits;        /* which metrics we are initializing */
struct rt_metrics rtm_rmx; /* metrics themselves */
```

The **if_msghdr** structure contains the following members:

```
uint16_t ifm_msglen;       /* to skip over unrecognized messages */
uint8_t  ifm_version;      /* future binary compatibility */
uint8_t  ifm_type;         /* message type */
int32_t  ifm_addrs;        /* bitmask identifying sockaddrs in
                            * the message */
int32_t  ifm_flags;        /* value of if_flags */
uint16_t ifm_index;        /* index for associated ifp */
struct if_data ifm_data;   /* statistics and other data about
                            * interface */
```

The **ifa_msghdr** structure contains the following members:

```
uint16_t ifam_msglen;      /* to skip over unrecognized messages */
uint8_t  ifam_version;     /* future binary compatibility */
uint8_t  ifam_type;        /* message type */
int32_t  ifam_addrs;       /* bitmask identifying sockaddrs in
                            * the message */
int32_t  ifam_flags;       /* value of ifa_flags */
uint16_t ifam_index;       /* index for associated ifp */
int32_t  ifam_metric;      /* value of ifa_metric */
```

To determine retransmission behavior, reliable protocols use the **rt_metrics** structure included in the **rt_msghdr** message header. The **rt_metrics** structure contains the following members:

```
uint32_t rmx_locks;        /* Kernel must leave these values alone */
uint32_t rmx_mtu;          /* MTU for this path */
uint32_t rmx_hopcount;     /* max hops expected */
uint32_t rmx_expire;       /* lifetime for route, e.g. redirect */
uint32_t rmx_recvpipe;     /* inbound delay-bandwidth product */
uint32_t rmx_sendpipe;     /* outbound delay-bandwidth product */
uint32_t rmx_ssthresh;     /* outbound gateway buffer limit */
uint32_t rmx_rtt;          /* estimated round trip time */
uint32_t rmx_rttvar;       /* estimated rtt variance */
uint32_t rmx_pksent;       /* packets sent using this route */
```

The **if_data** structure included in the **if_msghdr** message header defines a queue for a network interface and contains the following members:

r

```
/* generic interface information */
uint8_t ifi_type;          /* ethernet, tokenring, etc */
uint8_t ifi_physical;      /* AUI, Thinnet, 10base-T, etc */
uint8_t ifi_addrlen;       /* media address length */
uint8_t ifi_hdrlen;        /* media header length */
uint8_t ifi_recvquota;     /* polling quota for receive intrs */
uint8_t ifi_xmitquota;     /* polling quota for xmit intrs */
uint32_t ifi_mtu;          /* maximum transmission unit */
uint32_t ifi_metric;       /* routing metric (external only) */
uint32_t ifi_baudrate;     /* linespeed */

/* volatile statistics */
uint32_t ifi_ipackets;     /* packets received on interface */
uint32_t ifi_ierrors;      /* input errors on interface */
uint32_t ifi_opackets;     /* packets sent on interface */
uint32_t ifi_oerrors;      /* output errors on interface */
uint32_t ifi_collisions;   /* collisions on csma interfaces */
uint32_t ifi_ibytes;       /* total number of octets received */
uint32_t ifi_obytes;       /* total number of octets sent */
uint32_t ifi_imcasts;      /* packets received via multicast */
uint32_t ifi_omcasts;      /* packets sent via multicast */
uint32_t ifi_iqdrops;      /* dropped on input, this interface */
uint32_t ifi_noproto;      /* destined for unsupported protocol */
uint32_t ifi_hwassist;     /* HW offload capabilities */
uint32_t ifi_unused;       /* XXX was ifi_xmittiming */
struct timeval ifi_lastchange; /* time of last administrative change */
```

(Note that the position of items in all previously mentioned data structures does not necessarily reflect the order of the members in the structure.)

The members **rtm_addrs**, **ifm_addrs**, and **ifam_addrs** of the message headers are bitmasks that specify what socket address structure(s) follow the message. When multiple *sockaddrs* follow the message, they are interpreted based on their order in the message and the value stored in the bitmask. The sequence is least significant to the most significant bit within the vector.

The following constants are defined to indicate which socket addresses are present in the routing message:

```
#define RTA_DST     0x01   /* destination sockaddr present */
#define RTA_GATEWAY 0x02   /* gateway sockaddr present */
#define RTA_NETMASK 0x04   /* netmask sockaddr present */
#define RTA_GENMASK 0x08   /* cloning mask sockaddr present */
#define RTA_IFP     0x10   /* interface name sockaddr present */
#define RTA_IFA     0x20   /* interface address sockaddr present */
#define RTA_AUTHOR  0x40   /* author of redirect sockaddr present */
#define RTA_BRD     0x80   /* for NEWADDR, broadcast or
                            * point-to-point destination
                            * address */
```

Any messages sent to the kernel are returned back to the process issuing the command, and message copies are sent to all interested listeners. The sender may provide its process ID to be stored in the message header. An additional sequence field can be used to distinguish between outstanding messages. However, message replies may be lost when kernel buffers are exhausted.

Any messages generated by the kernel would have process ID and sequence field set to zero.

The kernel may spontaneously emit routing messages in response to external events, such as receipt of a redirect command, or failure to locate an appropriate route for a request. A process may ignore all messages from the routing socket by doing a *shutdown*(2) system call for further input.

Security Restrictions

Only users with appropriate privileges can make changes to the routing table.

Notes

Some fields in the message header structures are not used on HP-UX. This means when the kernel generates routing messages it sets these fields to 0. Also, when the kernel receives routing messages, it ignores any values contained in these fields. This applies to the following fields:

r

Structure	**Fields Not Used**
rt_msghdr	rtm_use, rtm_inits, rtm_rmx, except for rtm_rmx.rmx_mtu and rtm_rmx.rmx_rtt
if_msghdr	ifm_data, except for ifm_data.ifi_mtu, ifm_data.ifi_metric, ifm_data.ifi_ipackets, and ifm_data.ifi_opackets
ifa_msghdr	ifam_metric

ERRORS

If the kernel rejects a routing message, the **rtm_errno** field in the **rt_msghdr** structure may be set to one of the following values:

[EEXIST] The specified entry already exist. Requested to duplicate an existing entry.

[ENETUNREACH] Network is unreachable.

[ENOENT] The specified entry does not exist. Requested to delete non-existent entry.

[ENOBUFS] No buffer space is available. Insufficient resources were available to install a new route.

[EOPNOTSUPP] Operation not supported. Message types RTM_CHANGE and RTM_LOCK are not supported on HP-UX.

[EPERM] Permission to issue a command is denied. The user needs appropriate privileges to make changes to the routing table.

EXAMPLES

The following sample program illustrates how a user process can add a route to the kernel's routing table.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <net/route.h>
#include <net/if.h>
#include <netinet/in.h>

int main(int argc, char **argv)
{
        int    s;
        char   buf[1024];
        struct rt_msghdr *rtm;
        struct sockaddr_in *sin1, *sin2;

        if (argc != 3) {
                printf("usage: %s <destinationIP> <gatewayIP>\n",
                        argv[0]);
                return -1;
        }

        if ((s = socket(AF_ROUTE, SOCK_RAW, AF_UNSPEC)) < 0) {
                 perror("failed to create socket");
                 return -1;
        }

        rtm = (struct rt_msghdr *)buf;

        rtm->rtm_msglen = sizeof(struct rt_msghdr) +
                          (2 * sizeof(struct sockaddr_in));
        rtm->rtm_version = RTM_VERSION;
        rtm->rtm_type = RTM_ADD;
        rtm->rtm_addrs = (RTA_DST | RTA_GATEWAY);
        rtm->rtm_rmx.rmx_hopcount = 1;
        rtm->rtm_pid = getpid();
        rtm->rtm_errno = 0;
        rtm->rtm_seq = 0001;
```

```
                /*
                 * the destination address being added follows
                 * the routing header
                 */
                sin1 = (struct sockaddr_in *)(rtm + 1);
                sin1->sin_family = AF_INET;
                sin1->sin_addr.s_addr = inet_addr(argv[1]);

                /*
                 * the gateway address being added follows the
                 * destination address
                 */
                sin2 = (struct sockaddr_in *)(sin1 + 1);
                sin2->sin_family = AF_INET;
                sin2->sin_addr.s_addr = inet_addr(argv[2]);

                if (write(s, (caddr_t)rtm, rtm->rtm_msglen) < 0) {
                        perror("Failed to send routing message");
                        return -1;
                }

                return 0;
        }
```

AUTHOR

Routing socket interface was developed by HP and the University of California, Berkeley.

SEE ALSO

route(1M), ioctl(2), shutdown(2), socket(2), routing(7).

NAME

routing - system support for local network packet routing

DESCRIPTION

The network facilities for HP-UX provide general packet routing support. Routing table maintenance is handled by application processes.

A routing table consists of a set of data structures used by the network facilities to select the appropriate remote host or gateway when transmitting packets. The table contains a single entry for each route to a specific network or host, as displayed by the **netstat** command with the **-r** or **-rn** options (see *netstat*(1)). Routes that are not valid are not displayed.

```
# netstat -r
Routing tables
Destination      Gateway          Flags  Refs  Use Interface Pmtu
hpindwr.cup.hp.com
                 localhost        UH        1   39 lo0       4608
localhost        localhost        UH        0   68 lo0       4608
147.253.56.195   localhost        UH        0    0 lo0       4608
147.253.144.66   localhost        UH        0    0 lo0       4608
default          hpinsmh.cup.hp.com
                                  UG        1   21 lan0      1500
15.13.136        hpindwr.cup.hp.com
                                  U         1   92 lan0      1500
147.253.56       147.253.56.195   U         0    7 lan2      1500
147.253.144.64   147.253.144.66   U         0    7 lan1      1500
```

```
# netstat -rn
Routing tables
Destination      Gateway          Flags  Refs  Use Interface Pmtu
15.13.136.66     127.0.0.1        UH        1   39 lo0       4608
127.0.0.1        127.0.0.1        UH        0   68 lo0       4608
147.253.56.195   127.0.0.1        UH        0    0 lo0       4608
147.253.144.66   127.0.0.1        UH        0    0 lo0       4608
default          15.13.136.11     UG        2   30 lan0      1500
15.13.136.0      15.13.136.66     U         1  113 lan0      1500
147.253.56.0     147.253.56.195   U         0    7 lan2      1500
147.253.144.64   147.253.144.66   U         0    7 lan1      1500
```

```
# netstat -rv
Routing tables
Dest/Netmask     Gateway          Flags  Refs  Use Interface Pmtu
hpindwr.cup.hp.com/0xffffffff
                 localhost        UH        1   39 lo0       4608
localhost/0xffffffff
                 localhost        UH        0   68 lo0       4608
147.253.56.195/0xffffffff
                 localhost        UH        0    0 lo0       4608
147.253.144.66/0xffffffff
                 localhost        UH        0    0 lo0       4608
default/0x00000000
                 hpinsmh.cup.hp.com
                                  UG        2   31 lan0      1500
15.13.136/0xfffff800
                 hpindwr.cup.hp.com
                                  U         1  129 lan0      1500
147.253.56/0xfffffe00
                 147.253.56.195   U         0    7 lan2      1500
147.253.144.64/0xfffffff0
                 147.253.144.66   U         0    7 lan1      1500
```

r

```
# netstat -rnv
Routing tables
Dest/Netmask       Gateway          Flags  Refs  Use Interface Pmtu
15.13.136.66/255.255.255.255
                   127.0.0.1        UH        1   39 lo0       4608
127.0.0.1/255.255.255.255
                   127.0.0.1        UH        0   68 lo0       4608
147.253.56.195/255.255.255.255
                   127.0.0.1        UH        0    0 lo0       4608
147.253.144.66/255.255.255.255
                   127.0.0.1        UH        0    0 lo0       4608
default/0.0.0.0 15.13.136.11        UG        3   40 lan0      1500
15.13.136.0/255.255.248.0
                   15.13.136.66     U         1  153 lan0      1500
147.253.56.0/255.255.254.0
                   147.253.56.195   U         0    8 lan2      1500
147.253.144.64/255.255.255.240
                   147.253.144.66   U         0    8 lan1      1500
```

The following columns are of particular interest:

Destination The destination Internet address: host name, network name, or **default**. The **default** keyword indicates a wildcard route, used as a last resort if no route is specified for a particular remote host or network. See **Flags**.

Netmask The netmask and the destination Internet address together define a range of IP addresses that may be reached by the route's gateway. A host route by default has a netmask of all 1's. A default route by default has a netmask of all 0's. The netmask is also used in selecting a route to forward an IP packet. See the *Routing Algorithm* subsection.

Gateway The gateway to use to get to the destination: a remote gateway or the local host. See **Flags**.

Flags The type of route:

- **U** The route is "up" or available (see *ifconfig*(1M)).
- **G** The route uses a remote host as a gateway; otherwise, the local host is shown as the gateway (see *route*(1M)).
- **H** The destination is a host; otherwise, the destination is a network (see *route*(1M)).

Interface The interface connections:

- **lo0** The local loopback after system boot.
- **lan0**, **lan1**,... The interface cards installed on the local host after the **ifconfig** command is executed at boot time (see *ifconfig*(1M)).

The values of the *count* and *destination* type fields in the **route** command determine the presence of the **G** and **H** flags in the **netstat -r** display and thus the route type, as shown in the following table.

r

Count	Destination Type	Flags	Route Type
=0	network	**U**	Route to a network directly from the local host
>0	network	**UG**	Route to a network through a remote host gateway
=0	host	**UH**	Route to a remote host directly from the local host
>0	host	**UGH**	Route to a remote host through a remote host gateway
=0	**default**	**U**	Wildcard route directly from the local host
>0	**default**	**UG**	Wildcard route through a remote host gateway

Subnets

The network facilities support variable-length subnetting. An Internet address is made up of a **network address** portion, and a **host address** portion of an address in the form:

192.34.17.0

Subnet addresses are defined as a portion of the network's Internet address. This scheme provides for:

- Network addresses that identify physically distinct networks.
- Subnet addresses that identify physically distinct subnetworks of the same network.

A network manager can subdivide the Internet address of the local network into subnets using the host number space. This facility allows several physical networks to share a single Internet address.

To allow for this, three Internet classes are defined, each accommodating a different amount of network and host addresses. The address classes are defined by the most significant bit of the binary form of the address.

The following table lists the number of networks, nodes, and the address ranges for each address class:

Class	Networks	Nodes per Network	Address Range
A	127	16777215	0.0.0.1 - 127.225.225.254
B	16383	65535	128.0.0.1 - 191.255.255.254
C	2097151	255	192.0.0.1 - 223.244.244.243
Reserved	—	—	224.0.0.0 - 255.255.255.255

The first 8 bits of a Class A network has network space for only 127, while accommodating the largest number of nodes possible among the classes defined. A single class B network has the network address limitation of 16 bits, and 16 bits to define the nodes.

For example, a Class C address space is as follows:

```
Indicates                  Class C
Class C                    subnet
networks                   portion
   |                          |
  ---                        ---
  10000000.00000110.00000001.11100001
  --------------------------    -----
            |                     |
       Network Address           Host
          = 192.6.1             Address
                                 = 1
```

A subnet for a given host is specified with the **ifconfig** command (see *ifconfig*(1M)), using the **netmask** parameter with a 32-bit subnet *mask*.

The default masks for the three classes of Internet addresses are as follows:

Class A: 255.0.0.0
Class B: 255.255.0.0
Class C: 255.255.255.0

An example Class C network number is 192.34.17.0. The last field specifies the host number. Thus, all hosts with the prefix 192.34.17 are recognized as being on the same logical and physical network.

If subnets are not in use, the default mask used is 255.255.255.0.

If subnets are used and the 8-bit host field is partitioned into 3 bits of subnet and 5 bits of host as in the above example, then the subnet mask would be 255.255.255.192.

If a host has multiple interfaces, then it can belong to different subnets. Unlike past releases, the subnets can have different sizes even if they may have the same network address. This is accomplished by using a different netmask on each of the host interfaces. For example, the **lan1** and **lan2** interface shown in the **netstat** tables above are connected to two distinct subnets of the same network, 147.253. The subnet that **lan1** belongs to can have at most 14 hosts, because its netmask is 255.255.255.240.

Note:

The host portion of those IP addresses in the subnet cannot be all 1's or all 0's, therefore this subnet can support only 14 hosts, not 16.

The subnet that **lan2** belongs to can have up to 510 hosts, because its netmask is 255.255.254.0.

Supernets

A supernet is a collection of smaller networks. Supernetting is a technique of using the netmask to aggregate a collection of smaller networks into a supernet. This technique is particularly useful for class C networks. A Class C network can only have 254 hosts. This can be too restrictive for some companies. For these companies, a netmask that only contains a portion of the network part can be applied to the hosts in these class C networks to form a supernet. This supernet netmask should be applied to those interfaces that connect to the supernet using the *ifconfig* command (see *ifconfig*(1M)). For example, a host can configure its interface to connect to a class C supernet, for example, 192.6, by configuring an IP address of 192.6.1.1 and a netmask of 255.255.0.0 to its interface.

Routing Algorithm

The routing table entries are of three types:

- Entries for a specific host.
- Entries for all hosts on a specific network.
- Wildcard entries for any destination not matched by entries of the first two types.

To select a route for forwarding an IP packet, the network facilities select the complete set of "matching" routing table entries from the routing table. A routing table entry is considered a match, if the result of the bit-wise AND operation between the netmask in the routing entry and the IP packet's destination address equals the destination address in the routing entry.

The network facilities then select from the set the routing entries that have the longest netmask. The length of a netmask is defined as the number of contiguous 1 bits starting from the leftmost bit position in the 32-bit netmask field. In other words, the network facilities select the routing entry that specifies the narrowest range of IP addresses. For example, the host route entry that has a destination/netmask pair of (147.253.56.1, 0xFFFFFFFF), is more specific than the network route entry that has a destination/netmask pair of (147.253.56.0, 0xFFFFFE00); therefore, the network facilities select the host route entry. The default route by default has a destination/netmask pair of (0,0). Therefore, the default route matches all destinations but it is also the least specific. The default route will be selected only if there is not a more specific route.

There may still be multiple routing entries remaining. In that case, the IP packet is routed over the first entry displayed by **netstat -r**. Such multiple routes include:

- Two or more routes to a host via different gateways.
- Two or more routes to a network via different gateways.

A superuser can change entries in the table by using the **route** command (see *route*(1M), or by information received in Internet Control Message Protocol (ICMP) redirect messages.

If there are more than one default gateways for a particular net or subnet, each will be used in turn to effect the even distribution of datagrams to the different gateways.

WARNINGS

Reciprocal **route** commands must be executed on the local host and the destination host, as well as all intermediate hosts, if routing is to succeed in the cases of virtual circuit connections or bidirectional datagram transfers.

AUTHOR

routing was developed by the University of California, Berkeley.

FILES

/etc/hosts
/etc/networks

SEE ALSO

netstat(1), ifconfig(1M), route(1M), route(7P).

r

NAME

sad - STREAMS Administrative Driver

SYNOPSIS

#include <sys/types.h>
#include <sys/conf.h>
#include <sys/sad.h>
#include <stropts.h>

int ioctl(
    int *fildes***,**
    int *command***,**
    ...
    /* *arg* ***/**
    );

DESCRIPTION

The **sad** driver provides an interface to the **autopush** facility using the **ioctl()** function. As an interface, the **sad** driver enables administrative tasks to be performed on STREAMS modules and drivers. By specifying the *command* parameter to the **ioctl()** function, an administrator can configure **autopush** information for a device, get information on a device, or check a list of modules.

fildes is a file descriptor obtained by opening **/dev/sad** using **open()**. *command* specifies the administrative function to be performed. *arg* points to a data structure. If *command* is **SAD_SAP** or **SAD_GAP**, *arg* points to a struct of type **strapush**. If *command* is **SAD_VML**, *arg* points to a struct of type **str_list**.

Security Restrictions

The **SAD_SAP ioctl()** is restricted to superusers or users with the **NETADMIN** privilege. See *privileges*(5) for more information about privileged access on systems that support fine-grained privileges.

ioctl Commands

The commands used to perform administrative functions on a STREAMS module or driver are specified by the following **ioctl()** commands:

SAD_SAP

Allows you to configure **autopush** information for a device. The *arg* parameter points to a **strapush** structure (defined in the **<sys/sad.h>** header file), whose members are as follows:

```
struct strapush {
      uint sap_cmd;
      long sap_major;
      long sap_minor;
      long sap_lastminor;
      long sap_npush;
      char sap_list[MAXAPUSH][FMNAMESZ+1];
};
```

sap_cmd

Allows you to specify the type of configuration to perform. This field can have the following values:

SAP_ALL

Configures all minor devices.

SAP_RANGE

Configures a range of minor devices.

SAP_ONE

Configures a single minor device.

SAP_CLEAR

Clears the previous settings. Specify only the **sap_major** and **sap_minor** fields when using this command. If a previous entry specified

S

SAP_ALL, set the **sap_minor** field to 0 (zero). If a previous entry was specified as **SAP_RANGE**, set the **sap_minor** field to the lowest minor device number in the range.

sap_major
Specifies the major device number.

sap_minor
Specifies the minor device number.

sap_lastminor
Specifies the range of minor devices.

sap_npush
Specifies the number of modules to push. This number must be no more than **MAXAPUSH**, which is defined in **<sad.h>**. Additionally, this number must not exceed **NSTRPUSH**.

sap_list
Specifies, in order, the array of modules to push.

SAD_GAP
Lets you use the **sad** driver to obtain **autopush** configuration information for a device by setting the **sap_major** and **sap_minor** fields of the **strapush** structure (see the **SAD_SAP** command) to the major and minor device numbers of the device being queried.

arg should point to a struct of type **strapush**. Upon successful completion, the **strapush** structure contains all of the information used to configure the device. Values of 0 (zero) will appear in any unused entry in the module list.

SAD_VML
Enables you to check a list of modules. For example, you can determine if a specific module has been installed. The *arg* parameter points to a **str_list** structure (defined in the **<stropts.h>** header file), whose members are as follows:

```
struct str_list {
      int sl_nmods;
      struct str_mlist *sl_modlist;
};
```

sl_nmods
Specifies the number of entries you have allocated in an array.

sl_modlist
Points to the array of module names. The **str_mlist** structure (also in the **<stropts.h>** header file) is as follows:

```
struct str_mlist {
      char    l_name[FMNAMESZ+1];
};
```

where **l_name** specifies the array of module names.

If the **l_name** array is valid, the **SAD_VML** command returns a value of 0 (zero). If the array contains an invalid module name, the command returns a value of 1. Upon failure, the command returns a value of -1.

S

Notes

As a STREAMS driver, **sad** also supports the normal STREAMS **I_STR ioctl()**:

```
int ioctl(fildes, I_STR, strp);
int fildes;
struct strioctl *strp;
```

In this form, specify the **ic_cmd** field in the **strioctl** structure to either **SAD_SAP**, **SAD_GAP**, or **SAD_VML**. The **ic_dp** field points to the **strapush** structure (see the **SAD_SAP** command in the *DESCRIPTION* section). Refer to the *streamio*(7) reference page for further details.

RETURN VALUE

Unless specified otherwise, upon successful completion, the **sad ioctl()** commands return a value of 0 (zero). Otherwise, a value of -1 is returned.

ERRORS

If any of the following conditions occur, the **sad ioctl** commands return the corresponding value:

SAD_SAP

[EEXIST] The specified major/minor device number pair (**sad_major/sad_minor**) has already been configured.

[EFAULT] The *arg* parameter points outside the allocated address space.

[EINVAL] The major device number (**sad_major**) is invalid, the number of modules (**sap_list[MAXAPUSH][FMNAMESZ+1]**) is invalid, or the list of module names is invalid.

[ENODEV] The device is not configured for **autopush**. This value is returned from a **SAD_GAP** command.

[ENOSR] A internal **autopush** data structure cannot be allocated.

[ENOSTR] The major device does not represent a STREAMS driver.

[ERANGE] The **sap_lastminor** field is less than the **sap_minor** field when the command is **SAP_RANGE**, or the minor device specified in a **SAP_CLEAR** command does not exist.

[EACCES] Only a superuser or user with **NETADMIN** privilege is allowed to execute the **SAD_SAP ioctl()**.

SAD_GAP

[EFAULT] The *arg* parameter points outside the allocated address space.

[EINVAL] The major device number (**sad_major**) is invalid.

[ENODEV] The device is not configured for **autopush**.

[ENOSTR] The major device does not represent a STREAMS driver.

SAD_VML

[EFAULT] The *arg* parameter points outside the allocated address space.

[EINVAL] The list of module names is invalid.

SEE ALSO

autopush(1M), ioctl(2), open(2), privileges(5), streamio(7).

NAME

scsi - Small Computer System Interface device drivers

DESCRIPTION

The Small Computer System Interface (SCSI) is an American National Standard for interconnecting computers and peripheral devices. HP-UX supports the SCSI device protocol on parallel SCSI interfaces (see ANSI Std X3.131-199X, "SCSI-2"), Fibre Channel interfaces (see ANSI Std X3.269-199X, "Fibre Channel Protocol for SCSI"), and Serial Attached SCSI interfaces (SAS).

The SCSI standard includes specifications for a variety of device types. This section describes the general SCSI interface for all SCSI device drivers. Information about specific device types can be found in the manual sections which describe SCSI peripheral device drivers for those device types.

The ioctls described here can be issued either on persistent device files or legacy devices (see *intro*(7)). Legacy device files are deprecated with HP-UX release 11i V3. They are maintained for backward compatibility, and may be obsolete in future releases.

The behavior of some ioctls may differ depending on whether issued on persistent device files or legacy device files, and whether multi-pathing is enabled on legacy device files. Typically ioctls issuing SCSI commands to a device may use any available LUN path to send the commands. However, when multi-pathing is disabled on legacy device files (see **leg_mpath_enable** attribute in *scsimgr*(1M)), the ioctl only attempts to use the LUN path corresponding to the legacy device file. If this LUN path is not available, the ioctl will fail even if there are other LUN paths available. This behavior corresponds to the legacy behavior.

The **SIOC_INQUIRY** ioctl is supported by all SCSI device drivers. This ioctl returns the SCSI device-specific INQUIRY command data. This data contains device identification and capability information. Since there have been multiple versions of the SCSI standard for inquiry data, multiple versions of the inquiry data declaration are provided. The SCSI-1 version is provided for backward compatibility only. If issued on a legacy device file, this ioctl only tries to use the LUN path corresponding to the legacy device file even if multi-pathing is enabled on legacy device files.

The **SIOC_CAPACITY** ioctl indicates the current device size. A device size is defined to be a logical block size and some number of logical blocks. The means of determining this device-size data is particular to the specific device type. Logical block size and/or number of logical blocks equal to zero indicates: the device size is unknown, the device is not currently capable of I/O operations, or I/O operations are not meaningful for the device. Note that for very large devices, the ioctl argument can overflow, **SIOC_STORAGE_CAPACITY** is a better choice, than **SIOC_CAPACITY** where devices can be large. Also note that **DIOC_CAPACITY** is preferred (see *disk*(7)).

The header file **<sys/scsi.h>** has useful information for SCSI devices. The following is included from **<sys/scsi.h>**:

```
#define SIOC_INQUIRY            _IOR('S', 2, union inquiry_data)
#define SIOC_CAPACITY           _IOR('S', 3, struct capacity)
#define SIOC_STORAGE_CAPACITY   _IOR('S', 101, storage_capacity_t)

/* SCSI-1 inquiry structure */
struct inquiry {
        unsigned char    dev_type;
        unsigned int     rmb:1;
        unsigned int     dtq:7;
        unsigned int     iso:2;
        unsigned int     ecma:3;
        unsigned int     ansi:3;
        unsigned int     resv:4;
        unsigned int     rdf:4;
        unsigned char    added_len;
        unsigned char    dev_class[3];
        char             vendor_id[8];
        char             product_id[16];
        char             rev_num[4];
        unsigned char    vendor_spec[20];
        unsigned char    resv4[40];
        unsigned char    vendor_parm_bytes[32];
};
```

S

```
/* SCSI-2 inquiry structure */
struct inquiry_2 {
        unsigned int     periph_qualifier:3;
        unsigned int     dev_type:5;
        unsigned int     rmb:1;
        unsigned int     dtq:7;
        unsigned int     iso:2;
        unsigned int     ecma:3;
        unsigned int     ansi:3;
        unsigned int     aenc:1;
        unsigned int     trmiop:1;
        unsigned int     resv1:2;
        unsigned int     rdf:4;
        unsigned char    added_len;
        unsigned char    resv2[2];
        unsigned int     reladr:1;
        unsigned int     wbus32:1;
        unsigned int     wbus16:1;
        unsigned int     sync:1;
        unsigned int     linked:1;
        unsigned int     resv3:1;
        unsigned int     cmdque:1;
        unsigned int     sftre:1;
        char             vendor_id[8];
        char             product_id[16];
        char             rev_num[4];
        unsigned char    vendor_spec[20];
        unsigned char    resv4[40];
        unsigned char    vendor_parm_bytes[32];
} inquiry_2_t;

/* Definition for version description in SCSI-3 inquiry */
typedef uint8_t         vdesc_t[2];

/* SCSI-3 inquiry structure */
typedef struct inquiry_3 {
        uint32_t        pq              :3;
        uint32_t        pdt             :5;
        uint32_t        rmb             :1;
        uint32_t        rsvd1           :7;
        uint32_t        version         :8;
        uint32_t        aerc            :1;
        uint32_t        obslt1          :1;
        uint32_t        naca            :1;
        uint32_t        hisup           :1;
        uint32_t        rdf             :4;
        uint32_t        added_len       :8;
        uint32_t        sccs            :1;
        uint32_t        rsvd2           :7;
        uint32_t        bque            :1;
        uint32_t        encserv         :1;
        uint32_t        vs1             :1;
        uint32_t        multip          :1;
        uint32_t        mchngr          :1;
        uint32_t        obslt2          :1;
        uint32_t        obslt3          :1;
        uint32_t        addr16          :1;
        uint32_t        reladr          :1;
        uint32_t        obslt4          :1;
        uint32_t        wbus16          :1;
        uint32_t        sync            :1;
        uint32_t        linked          :1;
```

S

```
        uint32_t        obslt5          :1;
        uint32_t        cmdque          :1;
        uint32_t        vs2             :1;
        uint8_t         vendor_id[8];
        uint8_t         product_id[16];
        uint8_t         rev_num[4];
        uint8_t         vendor_spec[20];
        uint16_t        rsvd3           :4;
        uint16_t        clcking         :2;
        uint16_t        qas             :1;
        uint16_t        ius             :1;
        uint16_t        rsvd4           :8;
        vdesc_t         vers_desc[8];
        uint8_t         rsvd6[22];
        uint8_t         vendor_parm_bytes[32];
} inquiry_3_t;

/* union for SIOC_INQUIRY ioctl */
union inquiry_data {
        struct inquiry   inq1;    /* SCSI-1 inquiry */
        struct inquiry_2 inq2;    /* SCSI-2 inquiry */
        inquiry_3_t      inq3;    /* SCSI-3 inquiry */
};

/* structure for SIOC_CAPACITY ioctl */
struct capacity {
        uint32_t lba;
        uint32_t blksz;
};

/* structure for SIOC_STORAGE_CAPACITY ioctl */
typedef struct {
        uint64_t lba;
        uint32_t blksz;
} storage_capacity_t;
```

The **SIOC_XSENSE** ioctl returns detailed information about device status and errors when such information is available. Since there have been multiple versions of the SCSI standard for sense (status) data, multiple versions of the sense data declaration are provided. The SCSI-1 and non-aligned versions are provided for backward compatibility only. If no new CHECK-CONDITION-caused REQUEST SENSE command data has been obtained since the last **SIOC_XSENSE** ioctl call, the **xsense_aligned.error_class** and **sense_2_aligned.error_code** fields will contain the value zero. Applications which require more accurate REQUEST SENSE data handling should use the SCSI device-control driver (see *scsi_ctl*(7)).

S

The following information is included from **<sys/scsi.h>**:

```
#define SIOC_XSENSE          _IOR('S', 7, union sense_data)

/* structure for SIOC_XSENSE ioctl */
typedef union sense_data {
        xsense_aligned_t     r_sense1a;   /* SCSI and CCS devices */
        sense_2_aligned_t    r_sense2a;   /* SCSI-2 devices */
        xsense_t             r_sense1;    /* Do not use; for
                                           * compatibility only
                                           */
        sense_2_t            r_sense2;    /* Do not use; for
                                           * compatibility only
                                           */
} sense_data_t;

/*
 * Struct xsense_aligned is for examining the sense data of SCSI-1
 * and CCS devices.
```

```
 */
typedef struct xsense_aligned {
        unsigned int    valid           :1;
        unsigned int    error_class     :3;
        unsigned int    error_code      :4;
        unsigned char   seg_num;
        unsigned int    parms:4;
        unsigned int    sense_key       :4;
        unsigned char   lba[4];
        unsigned char   add_len;
        unsigned char   copysearch[4];  /* Unused by HP-UX */
        unsigned char   sense_code;
        unsigned char   resv;
        unsigned char   fru;
        unsigned char   field;
        unsigned char   field_ptr[2];
        unsigned char   dev_error[4];
        unsigned char   misc_bytes[106];
} xsense_aligned_t;

/*
 *  Struct sense_2_aligned is for examining the sense data
 *  of SCSI-2 devices
 */
typedef struct sense_2_aligned {
        unsigned int    info_valid      :1;
        unsigned int    error_code      :7;
        unsigned char   seg_num;
        unsigned int    filemark        :1;
        unsigned int    eom             :1;
        unsigned int    ili             :1;
        unsigned int    resv            :1;
        unsigned int    key             :4;
        unsigned char   info[4];
        unsigned char   add_len;
        unsigned char   cmd_info[4];
        unsigned char   code;
        unsigned char   qualifier;
        unsigned char   fru;
        unsigned char   key_specific[3];
        unsigned char   add_sense_bytes[113];
} sense_2_aligned_t;

/*
 * Struct xsense is provided for backward source code
 * compatibility only.
 * Struct xsense_aligned is the appropriate struct for
 * examining the sense
 * data of SCSI-1 and CCS devices.
 */
typedef struct xsense {
        unsigned int    valid           :1;
        unsigned int    error_class     :3;
        unsigned int    error_code      :4;
        unsigned char   seg_num;
        unsigned int    parms           :4;
        unsigned int    sense_key       :4;
        unsigned char   lba[4];
        unsigned char   add_len;
        unsigned char   copysearch[4];  /* Unused by HP-UX */
        unsigned char   sense_code;
        unsigned char   resv;
```

S

```
                unsigned char   fru;
                unsigned char   field;
                unsigned short  field_ptr;
                uint32_t        dev_error;
                unsigned char   misc_bytes[106];
        } xsense_t;

        /*
         * Struct sense_2 is provided for backward source code
         * compatibility only.
         * Struct sense_2_aligned is the appropriate struct for
         * examining the sense
         * data of SCSI-2 devices.
         */
        typedef struct sense_2 {
                unsigned int    info_valid      :1;
                unsigned int    error_code      :7;
                unsigned char   seg_num;
                unsigned int    filemark        :1;
                unsigned int    eom             :1;
                unsigned int    ili             :1;
                unsigned int    resv            :1;
                unsigned int    key             :4;
                unsigned char   info[4];
                unsigned char   add_len;
                unsigned int    cmd_info;
                unsigned char   code;
                unsigned char   qualifier;
                unsigned char   fru;
                unsigned char   key_specific[3];
                unsigned char   add_sense_bytes[113];
        } sense_2_t;
```

ERRORS

The following errors may result from a call to a SCSI device driver:

[EACCES] Required permission is denied for the device or operation.

[ENXIO] If resulting from an open call, this indicates there is no device at the specified address. For other calls, this indicates the specified address is out of range or the device may no longer be accessed.

[EINVAL] If resulting from an open call, this indicates the device is not supported by the device driver (e.g., incorrect device type). For other calls, this indicates the request or some request argument is invalid. If resulting from the **SIOC_CAPACITY** ioctl, one or more of the fields in the argument structure may have overflowed.

[EBUSY] This indicates the device is not ready for use or that the requested operation conflicts with other operations (e.g., the device is currently open via another device driver or exclusive access is in effect).

[EIO] Indicates a SCSI protocol or communication problem has occurred, or that a SCSI command resulted in a non-good status.

Manual entries that describe specific SCSI peripheral device drivers may provide additional qualification of error results.

WARNINGS

Use of devices that are not officially supported can cause data loss, system panics and device damage. HP-UX device drivers expect devices to be SCSI-2 compliant. Unsupported devices that are only SCSI-CCS compliant may work but their use is discouraged. Use of unsupported devices that are only SCSI-1 compliant is strongly discouraged.

Changing SCSI bus connectivity (recabling) while the system is running is not supported. Switching SCSI device power on or off while the device is connected to a system that does not support powerfail recovery is not supported. These activities are known to cause data loss and system panics.

On systems that support the **scsi_ctl** interface, the **SIOC_CMD_MODE**, **SIOC_SET_CMD**, and **SIOC_RETURN_STATUS** ioctls are obsolete (see *scsi_ctl*(7)). Direct manipulation of SCSI devices via the **scsi_ctl** interface provides a more functionally complete and easier-to-use means of low level SCSI device control (see *scsi_ctl*(7)).

Drivers that support only devices which have no meaningful size may not support the **SIOC_CAPACITY** ioctl. Total device size in bytes may exceed $2^{32}-1$ for some devices.

DEPENDENCIES

esdisk/estape/eschgr/sdisk/schgr/stape

The **SIOC_EXCLUSIVE** ioctl may be used to obtain and release exclusive access. Exclusive access, which prevents simultaneous access by other applications, is required for some operations and may be desirable in other circumstances. The following exclusive access control arguments are supported. The corresponding values are defined in **<sys/scsi.h>** If the ioctl is issued on a persistent device file, target and bus exclusive access actually result to LUN exclusive access.

SIOC_REL_LUN_EXCL	Release exclusive access to logical unit (LUN).
SIOC_SET_LUN_EXCL	Gain exclusive access to logical unit (LUN).
SIOC_REL_TGT_EXCL	Release exclusive access to associated SCSI target.
SIOC_SET_TGT_EXCL	Gain exclusive access to associated SCSI target.
SIOC_REL_BUS_EXCL	Release exclusive access to associated SCSI bus.
SIOC_SET_BUS_EXCL	Gain exclusive access to associated SCSI bus.

The **SIOC_MEDIUM_CHANGED** ioctl indicates when the media in a removable-media device may have changed. A value of "1" indicates the device media may have changed since the last **SIOC_MEDIUM_CHANGED** ioctl call. Note that only the first such call after a media change receives this indication. This means that media changes are likely to be missed if multiple applications are attempting to detect media changes. Exclusive access, obtained through use of the **SIOC_EXCLUSIVE** ioctl, can be used to avoid this problem.

The following information is included from **<sys/scsi.h>**:

```
#define SIOC_MEDIUM_CHANGED  _IOR('S', 42, int)
#define SIOC_EXCLUSIVE       _IOR('S', 68, int)
```

disc3

The **SIOC_VPD_INQUIRY** ioctl allows access to detailed device specific information. The **page_code** field specifies which SCSI vital product data page is requested. The **page_buf** field is filled with the requested page data. This ioctl when issued on a legacy device file only attempts to send the INQUIRY command through the LUN path corresponding to the legacy device file even if multi-pathing is enabled on legacy device files.

The following information is included from **<sys/scsi.h>**:

```
#define SIOC_VPD_INQUIRY  _IOWR('S', 10, struct vpd_inquiry)

/* union for SIOC_VPD_INQUIRY ioctl */
struct vpd_inquiry {
        char    page_code;       /* VPD page code           */
        char    page_buf[126];   /* buffer for VPD page info  */
};
```

FILES

/usr/include/sys/scsi.h

SEE ALSO

diskinfo(1M), ioctl(2), autochanger(7), intro(7), scsi_ctl(7), scsi_disk(7), scsi_tape(7).

NAME

scsi_ctl - SCSI pass-through driver (esctl/sctl)

DESCRIPTION

SCSI devices are controlled by a device-specific driver, when one exists. Device-specific drivers, such as those for SCSI direct access (disk) and sequential access (tape) devices, coordinate device and driver states to accomplish correct logical device behavior. The SCSI pass-through driver enables use of SCSI devices and commands not normally supported by these device-specific drivers.

esctl is the SCSI pass-through driver and works with persistent device files (see *intro*(7)). **sctl** is the SCSI pass-through driver already used on HP-UX releases prior to HP-UX 11i V3. It is maintained here for backward compatibility, and works with legacy device files. In this document **scsi_ctl** refers to both **esctl** and **sctl**.

Once the device is opened through **scsi_ctl** driver, ioctl calls can be used to change SCSI communication parameters or attempt SCSI commands and other SCSI operations. Since pass-through driver does not attempt to logically understand the target device, **read()** and **write()** calls are not supported.

Except where noted, the ioctls described here are available through all SCSI device drivers (including device-specific drivers). All **reserved** fields in the data structures associated with these ioctls must be zero-filled.

The following ioctls which are specific to parallel SCSI, are deprecated for issuance on LUN device special files (DSF). They are not supported on persistent device special files. They continue to be supported on legacy device special files for backward compatibility. But, it is recommended now to issue them or equivalent ioctls introduced with HP-UX 11i V3, directly on the parallel SCSI HBA device special file (DSF).

SIOC_GET_TGT_PARMS
SIOC_GET_BUS_PARMS
SIOC_GET_TGT_LIMITS
SIOC_GET_BUS_LIMITS
SIOC_SET_TGT_LIMITS
SIOC_SET_BUS_LIMITS

The following parallel SCSI specific ioctls introduced with HP-UX 11i V3 should be issued directly on the parallel SCSI HBA DSF. They replace some existing ioctls, which can no longer be issued on LUN persistent device files starting with HP-UX 11i V3:

PSIOC_GET_TGT_LIMITS	replaces **SIOC_GET_TGT_PARMS**
PSIOC_GET_TGT_PARMS	replaccs **SIOC_GET_BUS_PARMS**
PDIOC_RSTCLR	replaces **DIOC_RSTCLR**
PSIOC_RESET_DEV	replaces **SIOC_RESET_DEV**

Legacy device files are deprecated with HP-UX release 11i V3. They are maintained for backward compatibility, and may be obsolete in a future release (see *intro*(7) for details about legacy device file and persistent device files). It is recommended to use persistent device files for new applications.

S

Most of the ioctls described here can be issued either on persistent device files or legacy device files. The behavior of some ioctls may differ depending on whether issued on persistent device files or legacy device files, and whether multi-pathing is enabled on legacy device files. Typically ioctls issuing SCSI commands to a device may use any available LUN path to the device to send the commands. However, when multipathing is disabled on legacy device files (see **leg_mpath_enable** attribute in *scsimgr*(1M)), the ioctl only attempts to use the LUN path corresponding to the legacy device file. If this LUN path is not available, the ioctl will fail even if there are other LUN paths available. This behavior corresponds to the legacy behavior.

Device Special File Minor Number

The pass-through driver (**esctl/sctl**) is the preferred method to perform the ioctls **SIOC_IO_EXT** (**esctl** only) and **SIOC_IO** ioctls, rather than going through a device-specific driver (such as **esdisk**). To do this, you must create the device special file for the pass-through driver. *mksf*(1M) is the recommended method to create a pass-through device file for **esctl**. To create a device file for the legacy pass-through driver **sctl**, use *mknod*(1M), substituting the values in the minor number as noted:

/usr/sbin/mknod *name* **c 203 0x***iitl***0o**

where component parts of the minor number are constructed as follows:

ii Two hexadecimal digits, identifying the controlling interface card by its "Instance" number. The Instance value is displayed in *ioscan*(1M) output, under column **I** for the "Interface" hardware type.

t One hexadecimal digit identifying the drive (target) address.

l One hexadecimal digit identifying the logical unit number (LUN) within the device.

0 Hexadecimal digit zero, for reserved portion of the minor of the minor number.

o Optional values as follows:

0 To perform Inquiry on open to to ensure the device exists (recommended); or

2 To inhibit Inquiry on open. Starting with HP-UX 11i V3, option 2 is deprecated. It is maintained for binary compatibility with existing applications already setting it. Inquiry command will actually be sent during open, regardless of this option being set or not to 2.

SCSI Communication Parameters

HP-UX supports the SCSI device protocol on parallel SCSI interfaces, Fibre Channel interfaces, and Serial Attached SCSI interfaces. The SCSI communication parameters described here might only apply to certain SCSI interfaces and are noted as such in the descriptions.

SCSI communication parameters control features related to communication for three different scope levels: bus (link), target, and logical unit number (LUN). Bus communication parameters apply to all targets connected to a specific bus. Target communication parameters apply to all LUNs associated with a specific target. LUN communication parameters apply to a specific LUN. SCSI communication parameters apply to all device drivers (both device-specific and **scsi_ctl**).

At power-up and after being reset, all parallel SCSI devices and hosts communicate using asynchronous data transfers. Asynchronous data transfers use request (REQ) and acknowledge (ACK) signaling. The strict ordering of REQ and ACK signaling simplifies the communication protocol but limits I/O performance. A SCSI target and host pair may agree to use synchronous data transfers to increase I/O performance.

Synchronous data transfers improve I/O performance by lessening the ordering requirements on REQs and ACKs. By allowing multiple outstanding REQs, signal propagation delays and temporary rate imbalances are better tolerated. To make use of synchronous data transfers, a SCSI target and host must negotiate to determine mutually acceptable maximum REQ-ACK-offset and data-transfer rate parameters.

The maximum REQ-ACK-offset parameter indicates the maximum allowable number of outstanding REQs. The value zero is used to indicate asynchronous data transfer. Other values indicate synchronous data transfer. The appropriate value is generally dependent on the size of the receive data FIFO. High values tend to improve data transfer rates. The maximum data-transfer rate parameter indicates the "burst" data transfer rate (minimum allowable time between successive synchronous data transfers). A SCSI synchronous data transfer request (SDTR) message, used to initiate the negotiation process, is associated with the processing of a SCSI command.

At power-up and after being reset, all parallel SCSI devices and hosts communicate using eight-bit data transfers. A SCSI target and host pair may agree to use sixteen-bit (wide) data transfers to increase I/O performance. To make use of wide data transfers, a SCSI target and host must negotiate to determine a mutually acceptable data transfer width parameter. A SCSI wide data transfer request (WDTR) message, used to initiate the negotiation process, is associated with the processing of a SCSI command.

Some SCSI devices are able to simultaneously manage multiple active commands. Such a device has a command queue that holds commands for processing. Command queuing can improve I/O performance by reducing the time spent by the device waiting for new commands from the host. Note that command queuing might not improve I/O performance substantially for devices that support "read-ahead" and "immediate-reporting" (see *scsi_disk*(7) and *scsi_tape*(7)). The SCSI device and host use command tags to correctly manage these multiple simultaneously active commands. At all times when command queuing is in effect, each active command being handled by a specific LUN has a unique command tag.

SCSI devices indicate their ability to support the special communication features described above in their SCSI **INQUIRY** command data. Normally the SCSI **INQUIRY** command data and negotiation protocols allow hosts and devices to determine the optimal communication parameters so that I/O performance is maximized.

The current operating communication parameters may be determined by use of the: **SIOC_GET_LUN_PARMS**, **PSIOC_GET_TGT_PARMS** (recommended) or **SIOC_GET_TGT_PARMS** (for backward compatibility), and **SIOC_GET_BUS_PARMS** ioctls.

S

Occasionally, it is desirable to limit SCSI communication parameters to work around a communication problem or to provide external insight in determining optimal parameters. SCSI communication parameter limit suggestions can be specified by use of the: **SIOC_SET_LUN_LIMITS**, **SIOC_SET_TGT_LIMITS**, and **SIOC_SET_BUS_LIMITS** ioctls.

Note that there might be substantial differences between specified communication parameter limit suggestions and the corresponding actual current communication parameters being used for communication. These differences are a result of device-specific driver capabilities, interface driver capabilities, interface hardware capabilities, device capabilities, delays due to the negotiation process, delays due to currently active commands, and delays due to commands waiting to be sent to devices. Note that communication parameter limit suggestions might not survive between **close()** and **open()** calls, when no SCSI device drivers (device-specific or **scsi_ctl**) have associated LUN(s) open.

The current SCSI communication parameter limit suggestions may be determined by use of the **SIOC_GET_LUN_LIMITS**, **SIOC_GET_TGT_LIMITS**, and **SIOC_GET_BUS_LIMITS** ioctls.

Logical unit communication parameters may be managed by use of the **SIOC_GET_LUN_PARMS**, **SIOC_SET_LUN_LIMITS**, and **SIOC_GET_LUN_LIMITS**, **SIOC_RESET_DEV**, **SIOC_RESET_BUS** ioctls.

The **SIOC_GET_LUN_PARMS** ioctl indicates the current LUN communication parameter values. The *max_q_depth* field indicates whether or not tagged queuing is enabled, and if enabled, the maximum number of simultaneously active commands allowed. When *max_q_depth* is zero, tagged queuing is disabled. When it is one, tags are being used but commands are still being serially processed. When it is greater than one, tags are being used and *max_q_depth* specifies the maximum number of simultaneously active commands allowed.

The **SIOC_SET_LUN_LIMITS** ioctl may be used to provide LUN communication parameter limit suggestions. The *max_q_depth* field indicates whether or not tagged queuing should be enabled, and if enabled, the maximum number of simultaneously active commands that should be allowed. The **SIOC_GET_LUN_LIMITS** ioctl indicates the current LUN communication parameter limit suggestions.

Target communication parameters may be managed by use of the **PSIOC_GET_TGT_PARMS** ioctl on any associated HBA DSF, or **SIOC_GET_TGT_PARMS**, **SIOC_SET_TGT_LIMITS**, and **SIOC_GET_TGT_LIMITS** ioctls to any associated LUN.

The **PSIOC_GET_TGT_PARMS** and **SIOC_GET_TGT_PARMS** ioctls indicate the current target communication parameter values. The *width*, *reqack_offset*, and *xfer_rate* fields indicate the currently negotiated data transfer parameters. When *width* is eight, narrow transfers are in effect. When it is sixteen, wide transfers are in effect. When *reqack_offset* is zero, asynchronous transfers are in effect and *xfer_rate* is meaningless. When *reqack_offset* is non-zero, synchronous transfers are in effect and the maximum "burst" data transfer rate is *xfer_rate* words per second, where the size of a word is as indicated in *width*.

The **SIOC_SET_TGT_LIMITS** ioctl specifies the target communication parameter limit suggestions. The *max_width* field specifies maximum bus width that should be used for data transfers. The *max_reqack_offset* field specifies the maximum number of outstanding REQs that should be attempted during data transfers. The *max_xfer_rate* field specifies the maximum "burst" data rate that should be allowed during synchronous data transfers. The **SIOC_GET_TGT_LIMITS** ioctl indicates the current target communication parameter limit suggestions. The *width*, *reqack_offset*, *xfer_rate*, *max_width*, *max_reqack_offset*, *max_xfer_rate* fields only apply to parallel SCSI.

Bus communication parameters may be managed by use of the **SIOC_GET_BUS_PARMS**, **SIOC_SET_BUS_LIMITS**, and **SIOC_GET_BUS_LIMITS** ioctls to any associated LUN.

The **SIOC_GET_BUS_PARMS** ioctl indicates the current bus communication parameter values. The *max_width* field indicates the maximum data transfer width that will be attempted for data transfers to any target device connected to the associated bus. The *max_reqack_offset* field indicates the maximum number of outstanding REQs that will be attempted during data transfers to any target device connected to the associated bus. The *max_xfer_rate* field indicates the maximum "burst" data transfer rate that will be attempted for data transfers to any target device connected to the associated bus.

The **SIOC_SET_BUS_LIMITS** ioctl specifies the bus communication parameter limit suggestions for targets connected to the associated bus. The *max_width* field specifies the suggested maximum data transfer width that should be attempted for data transfers to any target device connected to the associated bus. The *max_reqack_offset* field specifies the maximum number of outstanding REQs that should be attempted during data transfers to any target device connected to the associated bus. The *max_xfer_rate* field specifies the maximum synchronous "burst" data transfer rate that should be attempted for data transfers to any target device connected to the associated bus. The **SIOC_GET_BUS_LIMITS** ioctl indicates the current bus communication parameter limit suggestions. The *max_width*, *max_reqack_offset*, and *max_xfer_rate* fields only apply to parallel SCSI.

The following is included from **<sys/scsi.h>**:

```
/* SCSI communication parameter ioctls */
#define SIOC_GET_LUN_PARMS      _IOR('S', 58, struct sioc_lun_parms)
#define SIOC_GET_TGT_PARMS      _IOR('S', 59, struct sioc_tgt_parms)
#define SIOC_GET_BUS_PARMS      _IOR('S', 60, struct sioc_bus_parms)
#define SIOC_GET_LUN_LIMITS     _IOR('S', 61, struct sioc_lun_limits)
#define SIOC_GET_TGT_LIMITS     _IOR('S', 62, struct sioc_tgt_limits)
#define SIOC_GET_BUS_LIMITS     _IOR('S', 63, struct sioc_bus_limits)
#define SIOC_SET_LUN_LIMITS     _IOW('S', 64, struct sioc_lun_limits)
#define SIOC_SET_TGT_LIMITS     _IOW('S', 65, struct sioc_tgt_limits)
#define SIOC_SET_BUS_LIMITS     _IOW('S', 66, struct sioc_bus_limits)

struct sioc_lun_parms {
        unsigned int flags;
        unsigned int max_q_depth;        /* maximum active I/O's */
        unsigned int reserved[4];        /* reserved for future
                                          * use
                                          */
} sioc_lun_parms_t;

struct sioc_lun_limits {
        unsigned int flags;
        unsigned int max_q_depth;
        unsigned int reserved[4];        /* reserved for
                                          * future use
                                          */
} sioc_lun_limits_t;

typedef struct sioc_tgt_parms {
        unsigned int flags;
        unsigned int width;             /* bits per word */
        unsigned int xfer_rate;         /* words per second */
        unsigned int reqack_offset;     /* REQ/ACK offset */
        unsigned int tgt_id;            /* target Id */
        unsigned int reserved[3];       /* reserved
                                         * for future use
                                         */
} sioc_tgt_parms_t;

typedef struct sioc_tgt_limits {
        unsigned int flags;
        unsigned int max_width;         /* Bits per word */
        unsigned int max_xfer_rate;     /* Words per second */
        unsigned int max_reqack_offset; /* REQ/ACK offset */
        unsigned int tgt_id;            /* target Id */
        unsigned int reserved[3];       /* Reserved for future
                                         * use
                                         */
} sioc_tgt_limits_t;

struct sioc_bus_parms {
        unsigned int flags;              /* reserved for future
                                          * use
                                          */
        unsigned int max_width;
        unsigned int max_reqack_offset;
        unsigned int max_xfer_rate;      /* bytes/sec */
        unsigned int reserved[4];        /* reserved for future
                                          * use
```

S

```
                                                */
    } sioc_bus_parms_t;

    struct sioc_bus_limits {
            unsigned int flags;              /* reserved for future
                                              *  use
                                              */
            unsigned int max_width;
            unsigned int max_reqack_offset;
            unsigned int max_xfer_rate;      /* bytes/sec */
            unsigned int reserved[4];        /* reserved for future
                                              * use
                                              */
    } sioc_bus_limits_t;
```

The following is included from **<sys/pscsi.h>**:

```
#define PSIOC_GET_TGT_PARMS   _IOWR('S', 114, struct sioc_tgt_parms)
#define PSIOC_GET_TGT_LIMITS  _IOWR('S', 115, struct sioc_tgt_limits)
#define PSIOC_RESET_DEV       _IOW('S', 116, int)
#define PDIOC_RSTCLR          _IOW('S', 117, int)
```

SCSI Commands and Operations

SIOC_IO_EXT and **SIOC_IO** ioctls allow an arbitrary SCSI command to be sent to a device. All details of the SCSI command protocol are handled automatically. **SIOC_IO_EXT** should only be issued on persistent device files. it allows to send the scsi command through any available LUN path or through a selected LUN path. **SIOC_IO** is deprecated. It can be issued on both persistent and legacy device files. When issued on a persistent device file, the SCSI command is sent through any available LUN path.

The following flags can be used to specify the *flags* field value of both **SIOC_IO_EXT** and **SIOC_IO**, unless indicated otherwise:

SCTL_READ — Data read operation is expected if *data_length* field is non-zero. The absence of this flag implies that data write operation is expected if the *data_length* field is non-zero.

SCTL_INIT_SDTR — Synchronous data transfer request negotiations should be attempted with this command. This flag only applies to parallel SCSI and is maintained for backward compatibility.

SCTL_INIT_WDTR — Wide data transfer request negotiations should be attempted with this command. This flag only applies to parallel SCSI and is maintained for backward compatibility.

SCTL_NO_DISC — discpriv bit in Identify message is not set. This flag only applies to parallel SCSI and is maintained for backward compatibility.

ESCTL_IO_LPT — The SCSI command is to be issued on a given LUN path. This flag can only be specified with **SIOC_IO_EXT** ioctl. When specified the hardware path of the LUN path to use is specified in field *lpt_hwp*

The *cdb* field specifies the SCSI command bytes. The number of command bytes is specified by the *cdb_length* field. These command bytes are sent to the target device during the SCSI command phase.

The address of the data area for the data phase of the SCSI command is specified by the *data* field. The *data_length* field specifies the maximum number of data bytes to be transferred. A zero-valued *data_length* indicates that no data phase should occur. Most SCSI commands with a data phase expect the data length information to be included somewhere in the command bytes. The caller is responsible for correctly specifying both the *data_length* field and any cdb data length values. The length may not be larger than **SCSI_MAXPHYS** and some implementations further restrict this length.

The *max_msecs* field specifies the maximum time, in milliseconds, that the device should need to complete the command. If this period of time expires without command completion, the system might attempt recovery procedures to regain the device's attention. These recovery procedures might include abort tag, abort, and device and bus reset operations. A zero value in the *max_msecs* field indicates that the timeout period is infinite and the system should wait indefinitely for command completion.

When the **SIO_IO_EXT** or **SIOC_IO** ioctl call returns, all command processing has been completed. Most **SIOC_IO_EXT**/**SIOC_IO** ioctl calls will return zero (success). The resulting detailed ioctl data

S

should be used to evaluate "success" or "failure" from the caller's perspective. The *cdb_status* field indicates the results of the **cdb** command. If the *cdb_status* field indicates a **S_CHECK_CONDITION** status, the *sense_status* field indicates the results of the SCSI **REQUEST SENSE** command used to collect the associated sense data. These status fields will contain one of the following values:

SCTL_INVALID_REQUEST	The SCSI command request is invalid and thus not attempted.
SCTL_SELECT_TIMEOUT	The target device does not answer to selection by the host SCSI interface (the device does not exist or does not respond).
SCTL_INCOMPLETE	The device answered selection but the command is not completed (the device took too long or a communication failure occurred).
S_GOOD	Device successfully completed the command.
S_CHECK_CONDITION	Device indicated sense data is available.
S_CONDITION_MET	Device successfully completed the command and the requested (search or pre-fetch) operation is satisfied.
S_BUSY	Device indicated it is unable to accept the command because it is busy doing other operations.
S_INTERMEDIATE	Device successfully completed this command, which is one in a series of linked commands (not supported, see *WARNINGS*).
S_I_CONDITION_MET	Device indicated both **S_INTERMEDIATE** and **S_CONDITION_MET** (not supported, see *WARNINGS*).
S_RESV_CONFLICT	Device indicated the command conflicted with an existing reservation.
S_COMMAND_TERMINATED	Device indicated the command is terminated early by the host system.
S_QUEUE_FULL	Device indicated it is unable to accept the command because its command queue is currently full.

The *data_xfer* field indicates the number of data bytes actually transferred during the data phase of the **cdb** command. This field is valid only when the *cdb_status* field contains one of the following values: **S_GOOD** or **S_CHECK_CONDITION**. The *sense_xfer* field indicates the number of valid sense data bytes. This field is valid only when the *cdb_status* field contains the value **S_CHECK_CONDITION** and the *sense_status* field contains the value **S_GOOD**.

The **SIOC_ABORT** ioctl causes a SCSI **ABORT** message to be sent to the LUN. This clears all active commands to the LUN from this initiator.

The **SIOC_TASK_MGMT** ioctl causes a SCSI task management function to be performed if supported by the SCSI transport. The following task management function values can be specified. They are defined in **<sys/scsi.h>**:

SIOC_TM_LUN_RESET	Lun Reset
SIOC_TM_WARM_TGT_RESET	Warm Target Reset
SIOC_TM_COLD_TGT_RESET	Cold Target Reset

The **SIOC_RESET_DEV** ioctl causes a SCSI device to be reset (including clearing all active commands). On parallel SCSI a **PSIOC_RESET_DEV** and **SIOC_RESET_DEV** ioctls cause a SCSI **BUS DEVICE RESET** message to be sent to the associated target. On Fibre Channel a **SIOC_RESET_DEV** ioctl causes a "TARGET RESET" task management function to be sent to the associated target followed by a Global Process Logout (GPRLO).

The **SIOC_RESET_BUS** ioctl causes the system to generate a SCSI bus reset condition on the associated bus. A SCSI bus reset condition causes all devices on the bus to be reset (including clearing all active commands on all devices). The **SIOC_RESET_BUS** ioctl does not apply to Fibre Channel.

Often it is necessary or useful to prohibit other SCSI commands while performing device-control operations. This should be done by gaining exclusive access via the **SIOC_EXCLUSIVE** ioctl. The associated argument points to an integer with one of these values defined in **<sys/scsi.h>**. Note that if the ioctl is issued on a persistent device file, target and bus exclusive access requests result to a LUN exclusive access being performed.

SIOC_REL_LUN_EXCL	release exclusive access to logical unit
SIOC_SET_LUN_EXCL	obtain exclusive access to logical unit

SIOC_REL_TGT_EXCL	release exclusive access to target
SIOC_SET_TGT_EXCL	obtain exclusive access to target
SIOC_REL_BUS_EXCL	release exclusive access to bus
SIOC_SET_BUS_EXCL	obtain exclusive access to bus

The ioctl **SIOC_PRIORITY_MODE** is deprecated with HP-UX release 11i V3. If called, it will just fake success. This ioctl was used to workaround situations where it is not possible to set exclusive access to the device. It put the device in "Priority mode". This caused all device-specific driver I/O operations (for example, file system I/O and virtual memory page swapping) and all SCSI device driver open calls (including pass-through driver open calls) to the associated LUN to block. These I/O operations and open calls were blocked for the entire duration that priority mode was in effect. While priority mode was in effect only **SIOC_IO** operations could be attempted. (these operations will not be blocked). It was very easy to cause system deadlock through incorrect use of the **SIOC_PRIORITY_MODE** ioctl. It normally required to lock the calling process into memory (see *plock*(2)) prior to enabling priority mode.

The header file **<sys/scsi.h>** has useful information for SCSI device control. The following is included from **<sys/scsi.h>**:

```
/* SCSI device control ioctls */
#define SIOC_IO             _IOWR('S', 22, struct sctl_io)
#define SIOC_RESET_DEV      _IO('S', 16)
#define SIOC_RESET_BUS      _IO('S', 9)
#define SIOC_PRIORITY_MODE  _IOW('S', 67, int)

#define SIOC_IO_EXT             _IOWR('S', 102, esctl_io_t)
#define SIOC_TASK_MGMT          _IOWR('S', 104, sioc_task_mgmt_t)

/* Structure for SIOC_IO_EXT ioctl */
typedef struct {
        int                     version;
        escsi_sctl_io_flags_t   flags;
        int                     max_msecs;
        uint32_t                cdb_length;
        uint32_t                data_length;
        ptr64_t                 data;
        union sense_data        sense;
        escsi_hw_path_t         lpt_hwp;
        uint32_t                data_xfer;
        uint32_t                sense_xfer;
        uint32_t                cdb_status;
        uint32_t                sense_status;
        uint8_t                 cdb[ESCSI_MAX_CDB_LEN];
        uint32_t                rsvd[32];  /* Reserved for
                                            * future use
                                            */
} esctl_io_t;

/* Structure for SIOC_IO ioctl */
struct sctl_io
{
        unsigned        flags;
        unsigned char   cdb_length;
        unsigned char   cdb[16];
        void            *data;
        unsigned        data_length;
        unsigned        max_msecs;
        unsigned        data_xfer;
        unsigned        cdb_status;
        unsigned char   sense[256];
        unsigned        sense_status;
        unsigned char   sense_xfer;
        unsigned char   reserved[64];
} sctl_io_t;
```

S

Security Restrictions

Superuser or **DEVOPS** privilege, or device write permissions are required to use these ioctls. See *privileges*(5) for more information about privileged access on systems that support fine-grained privileges.

EXAMPLES

Assume that *fildes* is a valid file descriptor for a persistent device file of a SCSI device, and *leg_fildes* is a valid file descriptor for a legacy device file of a SCSI device, and *lpt_hwp* contains a valid hardware path of a LUN path to the device. The first example attempts a SCSI **INQUIRY** command:

```
#include <sys/scsi.h>

esctl_io_t esctl_io;
#define MAX_LEN 255
unsigned char inquiry_data[MAX_LEN];

memset(&esctl_io, 0, sizeof(esctl_io)); /* clear reserved fields */
esctl_io.flags = SCTL_READ;   /* input data expected */
esctl_io.cdb[0] = CMDinquiry;
esctl_io.cdb[1] = 0x00;
esctl_io.cdb[2] = 0x00;
esctl_io.cdb[3] = 0x00;
esctl_io.cdb[4] = MAX_LEN;            /* allocation length */
esctl_io.cdb[5] = 0x00;
esctl_io.cdb_length = 6;              /* 6 byte command */
esctl_io.data = &inquiry_data[0];     /* data buffer location */
esctl_io.data_length = MAX_LEN;       /* maximum transfer length */
esctl_io.max_msecs = 10000;           /* allow 10 seconds for cmd */
if (ioctl(fildes, SIOC_IO_EXT, &esctl_io) < 0) {
        /* request is invalid */
} else {
        if ( esctl_io.cdb_status == S_GOOD) {
                /* success. display inquiry data */
        else {
                /* failure. process depending on cdb_status */
        }
}
```

The second example attempts a SCSI **INQUIRY** command via a specific LUN path.

```
#include <sys/scsi.h>

esctl_io_t esctl_io;
#define MAX_LEN 255
unsigned char inquiry_data[MAX_LEN];

memset(&esctl_io, 0, sizeof(esctl_io)); /* clear reserved fields */
esctl_io.flags = SCTL_READ | SCTL_IO_LPT;  /* input data
                                            * expected and commmand
                                            * to be sent on given
                                            * LUN path
                                            */
memcpy(&esctl_io.lpt_hwp, lpt_hwp, sizeof(lpt_hwp); /* specify
                                            * the hardware path of
                                            * LUN path through which
                                            * command must be sent
                                            */
esctl_io.cdb[0] = CMDinquiry;
esctl_io.cdb[1] = 0x00;
esctl_io.cdb[2] = 0x00;
esctl_io.cdb[3] = 0x00;
esctl_io.cdb[4] = MAX_LEN;            /* allocation length */
esctl_io.cdb[5] = 0x00;
esctl_io.cdb_length = 6;              /* 6 byte command */
esctl_io.data = &inquiry_data[0];     /* data buffer location */
```

S

```
esctl_io.data_length = MAX_LEN;       /* maximum transfer length */
esctl_io.max_msecs = 10000;           /* allow 10 seconds for cmd */
if (ioctl(fildes, SIOC_IO_EXT, &esctl_io) < 0) {
        /* request is invalid */
} else {
        if ( esctl_io.cdb_status == S_GOOD) {
                /* success. display inquiry data */
        else {
                /* failure. process depending on cdb_status */
        }
}
```

The following example attempts a SCSI **TEST UNIT READY** command and checks to see if the device is ready, not ready, or in some other state.

```
#include <sys/scsi.h>

struct sctl_io sctl_io;

memset(&sctl_io, 0, sizeof(sctl_io)); /* clear reserved fields */
sctl_io.flags = 0;                   /* no data transfer expected */
sctl_io.cdb[0] = 0x00;               /* can use CMDtest_unit_ready */
sctl_io.cdb[1] = 0x00;
sctl_io.cdb[2] = 0x00;
sctl_io.cdb[3] = 0x00;
sctl_io.cdb[4] = 0x00;
sctl_io.cdb[5] = 0x00;
sctl_io.cdb_length = 6;              /* 6 byte command */
sctl_io.data = NULL;                 /* no data buffer is provided */
sctl_io.data_length = 0;             /* do not transfer data */
sctl_io.max_msecs = 10000;           /* allow 10 seconds for cmd */
if (ioctl(leg_fildes, SIOC_IO, &sctl_io) < 0) {
        /* request is invalid */
} else if (sctl_io.cdb_status == S_GOOD) {
        /* device is ready */
} else if (sctl_io.cdb_status == S_BUSY ||
          (sctl_io.cdb_status == S_CHECK_CONDITION &&
           sctl_io.sense_status == S_GOOD &&
           sctl_io.sense_xfer > 2 &&
           (sctl_io.sense[2] & 0x0F) == 2)) {
        /* can use sense_data */
        /* device is not ready */
} else {
        /* unknown state */
}
```

S

WARNINGS

Incorrect use of **scsi_ctl** operations (even those attempting access to non-existent devices) can cause data loss, system panics, and device damage.

The **SIOC_EXCLUSIVE** ioctl should be used to gain exclusive access to a device prior to attempting **SIOC_IO** commands. If exclusive access is not obtained, **SIOC_IO** commands will be intermixed with device-specific driver commands, which can lead to undesirable results.

Device-specific drivers can reject inappropriate or troublesome **SIOC_IO_EXT/SIOC_IO** commands. However, since not all such operations are known and detected, care should be exercised to avoid disrupting device-specific drivers when using commands that modify internal device states.

Most SCSI commands have a logical unit number (LUN) field. Parallel SCSI implementations on the HP-UX operating system select logical units via the SCSI **IDENTIFY** message. The LUN portion of the cdb should normally be set to zero, even when the LUN being accessed is not zero.

Use of linked commands is not supported.

Most SCSI commands with a data phase expect the data length information to be included somewhere in the command bytes. Both the *data_length* field and any cdb data length values must be correctly specified to get correct command results.

Very large (or infinite) timeout values can cause a parallel SCSI bus (potentially the entire system) to hang.

Device and/or bus reset operations can be used to regain a device's attention when a timeout expires.

Resetting a device can cause I/O errors and/or loss of cached data. This can result in loss of data and/or system panics.

Obtaining SCSI **INQUIRY** data by use of the **SIOC_INQUIRY** ioctl instead of by use of the **SIOC_IO** ioctl is generally preferable since SCSI implementations on the HP-UX operating system synchronize access of inquiry data during driver open calls.

Since communication parameters can be affected by device-specific driver capabilities, device-specific driver use might result in communication parameter changes.

The **SIOC_CAPACITY** ioctl is not supported by **scsi_ctl** because the meaning of capacity is device-specific.

FILES

/usr/include/sys/scsi.h
/usr/include/sys/scsi_ctl.h

SEE ALSO

mknod(1M), mksf(1M), ioctl(2), plock(2). privileges(5), intro(7), scsi(7).

S

NAME

scsi_disk - SCSI direct access device drivers (esdisk/sdisk)

DESCRIPTION

This section describes the interface for access of SCSI disk, CD-ROM, and optical disk devices through the character special device driver. **esdisk** is the default driver for direct access devices starting at HP-UX 11i Version 3. **sdisk** is the default driver used on HP-UX 11i Version 2 and earlier releases. It is maintained for backward compatibility.

SCSI direct access devices store a sequence of data blocks. Each direct access device has a specific device size consisting of a number of data blocks and a logical block size. All data blocks have the same logical block size.

Since I/O operations must have a size that is an integral number of blocks, one logical block size is the smallest possible I/O quantity. The device block size can be determined through use of the **DIOC_DESCRIBE**, **DIOC_CAPACITY**, **SIOC_CAPACITY**, **DIOC_DESCRIBE_EXT**, and **SIOC_STORAGE_CAPACITY** ioctls (see *disk*(7) and *scsi*(7); **SIOC_CAPACITY** is not supported on **disc3**). A direct access device that is not ready for use, whether due to no media installed or another reason, is interpreted to mean the device has zero size. An **open()** call to such a device succeeds, but subsequent **read()** and **write()** calls fail.

The *ioctl*(2) manpage explains how the operations and arguments are used. Note, the *arg* used is commonly the address of the parameter cited in the particular ioctl **#define** statement. See the *EXAMPLES* section for sample code.

To improve performance, many SCSI disk devices have caches, which can be used for both read and write operations.

Read cache use, called "read ahead", causes the disk drive to read data in anticipation of read requests. Read ahead is only apparent to users in the increased performance that it produces.

Write cache use is called "immediate reporting". Immediate reporting increases I/O performance by reporting a completed write status before the data being written is actually committed to media. If the subsequent physical write operation does not complete successfully, data may be lost.

Physical write failures due to media defects are largely eliminated by use of automatic sparing in disk drives. Power failure between immediate reporting and media commit can result in cached data being lost. However, the period of time between these events is typically relatively small, making such losses unlikely.

The **SIOC_GET_IR** ioctl can be used to determine if immediate-reporting functionality is currently being used by the device. The value **1** indicates immediate reporting is enabled. The value zero indicates immediate reporting is disabled. The **SIOC_SET_IR** ioctl can be used to enable or disable immediate reporting. A zero value disables immediate reporting. The value **1** enables immediate reporting.

The **SIOC_SYNC_CACHE** ioctl can be used to force data cached in the device to media.

Most SCSI removable media disk devices support "prevent" and "allow" media-removal commands. To avoid data corruption and data accessibility problems, media removal is prevented for the entire duration a removable media disk device is open. Because media removal is not supported, the **SIOC_MEDIUM_CHANGED** ioctl is not supported.

The header file **<sys/scsi.h>** has useful information for direct access device control, including the following:

```
/* ioctl support for SCSI disk devices */
#define SIOC_GET_IR          _IOR('S', 14, int)
#define SIOC_SET_IR          _IOW('S', 15, int)
#define SIOC_SYNC_CACHE      -IOW('S', 70, int)
```

The **SIOC_FORMAT** ioctl reformats the entire media surface. Exclusive access to the device, obtained through use of the **DIOC_EXCLUSIVE** ioctl (see *disk*(7)), is required prior to reformatting to ensure that other applications are not affected. The **fmt_optn** field can be used to select the desired media geometry. Only one media geometry is supported on most devices. The value zero should be used for these devices. The value zero can also be used to select the default geometry on devices that support multiple media geometries. The interleave field can be used to specify sector interleaving. The value zero specifies that an appropriate default interleave should be used.

S

EXAMPLES

The following sample code shows how to use ioctls that affect **scsi_disk**.

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/errno.h>
#include <sys/diskio.h>
#include <sys/scsi.h>
Describe_ext(dfd)
  int dfd;
{
   int ret;
   disk_describe_type_ext_t disk_descr;
   uint64_t  capacity;

   if ((ret = ioctl (dfd, DIOC_DESCRIBE_EXT, &descr_type)) != 0) {
       exit(1);
   }
   printf("\nSuccessful ioctl DIOC_DESCRIBE_EXT \n");
   printf("  model number: %s\n", disk_descr.model_num);
   printf("  interface:    %d  <20=scsi>\n", disk_descr.intf_type);
   capacity = (disk_descr.maxsva_high << 32) + disk_descr.low_lba;
   printf("  Capacity:    %llu (blocks)\n", capacity);
   printf("  block size:  %u (bytes)\n", disk_descr.lgblksz);
   printf("  Device type:  %u (0=disk, 5=CD, 7=OM)\n",
             disk_descr.dev_type);
   printf("  Write Protected:  %s \n",
             disk_descr.flags & WRITE_PROTECT_FLAG ? "yes" : "No");

}

Describe (dfd)
  int dfd;
{
  int ret;
  disk_describe_type descr_type;
  if ((ret = ioctl (dfd, DIOC_DESCRIBE, &descr_type)) != 0) {
    exit(1);
  }
  printf ("\nSuccessful ioctl DIOC_DESCRIBE \n");
  printf ("  model number: %s\n", descr_type.model_num);
  printf ("  interface:    %d  <20=scsi>\n", descr_type.intf_type);
}
Exclusive (dfd)
  int dfd;
{
  int ret, flag=1;
  if ((ret = ioctl (dfd, DIOC_EXCLUSIVE, &flag)) != 0) {
    exit(1);
  }
}
Enable_WOE (dfd)
  int dfd;
{
  int ret, flag=1;
  if ((ret = ioctl (dfd, SIOC_WRITE_WOE, &flag)) != 0) {
    exit(1);
  }
  printf ("\nSuccessful ioctl SIOC_WRITE_WOE \n");
}
main (argc, argv)
  int argc;
```

S

```
  char ** argv;
{ int ret, fd; if (argc != 2) {
    printf ("Usage: %s <disk_device> \n", argv[0]);
    exit(1);
  }
  if ((fd = open (argv[1], O_RDWR)) < 0) {
    exit (1);
  }
  Describe_ext(fd);
  Describe (fd);
  Exclusive (fd);
  Enable_WOE (fd);
}
```

WARNINGS

Historically, disk devices have had small (typically 512 byte) block sizes; however, many newer disk devices (such as optical disks and disk arrays) have relatively large block sizes. Applications using direct raw disk access should use the **DIOC_DESCRIBE**, **DIOC_CAPACITY**, **DIOC_DESCRIBE_EXT**, or **SIOC_CAPACITY** ioctl to determine the appropriate minimum I/O size.

Media removal and insertion while a disk device is open is unsupported and unpredictable. Do not attempt to circumvent prevention of media removal. Device capacity changes resulting from such intervention may not be recognized.

Often larger I/O operation sizes are expected to be more efficient. However, SCSI disk I/O operations that are large relative to the device's cache can result in insufficient cache space for the device to maintain full-media-speed data transfer rates. This can result in decreased I/O performance relative to smaller I/O sizes.

DEPENDENCIES

Optical Disk Devices

The **SIOC_VERIFY_WRITES** ioctl controls the write mode. Normally written data is assumed to be correctly stored on the media. Verify-writes mode causes verification of written data to ensure that data has been correctly written. Verification can substantially reduce write performance and is not generally needed.

The **SIOC_VERIFY_WRITES** ioctl can be used to enable or disable write verification. A zero value disables write verification. The value **1** enables write verification. Although write verification is primarily intended for optical media, some systems may support write verification on normal disk devices.

The **SIOC_VERIFY** ioctl verifies that a media area contains valid data (that is, data that has been correctly written). Verified media will not cause I/O errors when reading is attempted. The media area to be verified is specified via the **start_lba** and **block_cnt** fields. Although verification is intended primarily for optical media, some systems may support verify operations on normal disk devices.

The **SIOC_WRITE_WOE** ioctl controls the write mode used for magneto-optical disk devices. Normally magneto-optical write operations require two physical head passes. The first pass erases the media area to be written. The second pass actually writes the data. Write-without-erase mode dramatically increases write performance by skipping the first (erase media area) pass. To ensure that the correct data results, it is essential that write-without-erase operations be performed only on media that is known to be blank (previously erased or never used). The **SIOC_WRITE_WOE** ioctl can be used to enable or disable write-without-erase. A zero value disables write-without-erase. The value **1** enables write-without-erase.

The **SIOC_ERASE** ioctl allows media areas to be explicitly erased. The media area to be erased is specified via the **start_lba** and **block_cnt** fields. Media areas erased in this manner can be written using write-without-erase mode. Note that an erased media area is different from a media area written with some data values (e.g. zeros). An erased media area should not be read. Attempting to read an erased media area generally results in an I/O error.

The **SIOC_VERIFY_BLANK** ioctl verifies that a media area has been erased and is suitable for being written using write-without-erase mode. The media area to be verified is specified via the **start_lba** and **block_cnt** fields.

The following optical disk device specific information is included from **<sys/scsi.h>**:

```
#define SIOC_WRITE_WOE        _IOW('S', 17, int)
#define SIOC_VERIFY_WRITES    _IOW('S', 18, int)
#define SIOC_ERASE            _IOW('S', 19, struct scsi_erase)
```

```
#define SIOC_VERIFY_BLANK     _IOW('S', 20, struct scsi_verify)
#define SIOC_VERIFY           _IOW('S', 21, struct scsi_verify)

/* structure for SIOC_ERASE ioctl */
struct scsi_erase {
        unsigned int    start_lba;
        unsigned short  block_cnt;
};

/* structure for SIOC_VERIFY_BLANK and SIOC_VERIFY ioctls */
struct scsi_verify {
        unsigned int    start_lba;
        unsigned short  block_cnt;
};
```

FILES

/usr/include/sys/scsi.h

SEE ALSO

mediainit(1), mknod(1M), ioctl(2), disk(7), scsi(7).

S

NAME

scsi_tape - SCSI sequential access device driver

DESCRIPTION

SCSI sequential-access (tape) devices store a sequence of data blocks. Data can be read and written using either fixed or variable sized block mode. If supported by the device, variable sized block mode is normally used (even when all blocks are the same size). Fixed sized block mode is generally only used for tape devices which do not support variable sized blocks. Fixed sized block mode can be used on some tape devices which support variable sized blocks to increase I/O performance.

Generally SCSI tape devices are controlled through the **mt** (see *mt*(7)) generic tape device interface. This section describes features that are specific to SCSI tape devices.

The **SIOC_CAPACITY** ioctl (see *scsi*(7)) can be used to determine remaining tape capacity for some tape devices. The **blksz** field indicates the "natural" block size of the device. This value may or may not be the current block size of the device. The number of blocks, indicated by the **lba** field, is an estimate of how much data can be written on the remaining media. A zero size is returned for devices that do not provide remaining-capacity information. The quantity of data that can actually be written may be higher or lower than indicated, depending on such factors as block size, media defects, data compression, and ability to maintain streaming.

To improve performance, most SCSI tape devices have caches. Read-cache use, called "read ahead", causes the tape drive to read data in anticipation of read requests. Read ahead is only apparent to users in the increased performance that it produces. Write-cache use is called "immediate reporting". Immediate reporting increases I/O performance by reporting a completed write status before the data being written is actually committed to media. This allows the application program to supply additional data so that continuous media motion, called "streaming", can be achieved. The **SIOC_GET_IR** ioctl can be used to determine if immediate-reporting functionality is currently being used by the device. The value "1" indicates immediate reporting is enabled. By default, the device driver attempts to enable immediate reporting. The **SIOC_SET_IR** ioctl can be used to explicitly enable or disable immediate reporting. A zero value disables immediate reporting. The value "1" enables immediate reporting. The **MTIOCTOP** ioctl **MTNOP** command can be used to cause any cached data to be written (committed) to media. Note that the device immediate reporting mode set by the **SIOC_SET_IR** ioctl survives between **close()** and **open()** calls, but not through system reboot.

The **SIOC_GET_BLOCK_SIZE** ioctl indicates the device's current block size. A block size of zero indicates the device is in variable-sized-block mode. A non-zero block size indicates the device is in fixed-sized-block mode.

The **SIOC_SET_BLOCK_SIZE** ioctl changes the current block size to the specified number of bytes. Setting the block size to zero specifies that variable-sized-block mode should be used. Any non-zero block size specifies that fixed-sized-block mode should be used. By default, the device driver attempts to set the block size to zero during open. If variable-sized-block mode is not supported by the device, the driver selects an appropriate block size for fixed-sized-block mode use. Note that the device block size set by the **SIOC_SET_BLOCK_SIZE** ioctl survives between **close()** and **open()** calls, but not through system reboot.

S

The **SIOC_GET_BLOCK_LIMITS** ioctl indicates the device's maximum and minimum fixed block-size limits. The device's minimum fixed block size is indicated by the **min_blk_size** field. The **max_blk_size** field contains the smaller of the maximum block size supported by the device and the maximum block size supported by the system (**MAXPHYS**). This is the largest valid block size for the specific combination of device, driver, and host system being used.

The **SIOC_GET_POSITION** ioctl can be used to determine the current media position for some devices. For devices that support this capability, the resultant value can be used to reposition the media to the same position in the future.

The **SIOC_SET_POSITION** ioctl can be used to cause media repositioning on some devices. For devices that support this capability, media repositioning via this mechanism can generally be completed more quickly than might be similarly accomplished using record, filemark, or setmark spacing. The argument value specified should be the result of a previous **SIOC_GET_POSITION** for that media volume.

The following is included from **<sys/scsi.h>**:

```
/* ioctl support for SCSI tape commands */
#define SIOC_GET_IR             _IOR('S', 14, int)
#define SIOC_SET_IR             _IOW('S', 15, int)
```

```
#define SIOC_GET_BLOCK_SIZE     _IOR('S', 30, int)
#define SIOC_SET_BLOCK_SIZE     _IOW('S', 31, int)
#define SIOC_GET_BLOCK_LIMITS   _IOW('S', 32, struct scsi_block_limits)
#define SIOC_GET_POSITION       _IOR('S', 33, int)
#define SIOC_SET_POSITION       _IOW('S', 34, int)

/* structure for SIOC_GET_BLOCK_LIMITS ioctl */
struct scsi_block_limits {
        unsigned min_blk_size;
        unsigned max_blk_size;
};
```

WARNINGS

SCSI bus and device resets cause some devices to reposition media to beginning-of-tape (BOT). This unintentional media repositioning can cause loss of data. The **scsi_tape** driver causes the first subsequent **open()** attempt to fail as an indication of potential data loss.

The **scsi_tape** driver does not write filemarks at close if the media has been programmatically repositioned. Applications that reposition the media prior to closing the device should write any required tapemarks.

SEE ALSO

mknod(1M), mt(7), scsi(7).

S

NAME

scsimgr_eschgr - SCSI class driver eschgr plug-in for scsimgr

DESCRIPTION

The SCSI class driver **eschgr** plug-in for **scsimgr** implements management and diagnostic operations specific to classes of devices bound to the **eschgr** driver. **eschgr** is the native HP-UX SCSI class driver that handles, by default, all library/changer devices.

The plug-in handles the following operations for driver **eschgr**:

- Display and clear driver **eschgr** global statistics and the statistics it maintains on instances of LUNs bound to it, and on related LUN paths.
- Display status and other information maintained by driver **eschgr** on LUNs bound to it.
- Get, set and save driver **eschgr** global and per-lun instance attributes.

Commands

The user can explicitly send the following **scsimgr** commands to driver **eschgr** plug-in by specifying the **-d eschgr** option:

clear_stat Clears statistics.

get_attr Displays information on attributes.

get_info Displays status and other information.

get_stat Displays statistics.

save_attr Saves value of attributes in a persistent store.

set_attr Set current values of attributes.

Note: Refer to *scsimgr*(1M) for the syntax of the above commands.

However, the only instances where it is necessary to explicitly send a command to the plug-in is when performing operations on objects global to driver **eschgr**: global statistics, attributes or status information. In all the other cases, **scsimgr** automatically invokes the plug-in to perform the driver specific part of the operation, when the operation applies to LUNs bound to driver **eschgr** or to their LUN paths.

Attributes

The following table lists driver **eschgr** specific attributes. For details on the concept of attribute refer to *scsimgr*(1M).

Note: The following conventions are used:

- RO is Read Only.
- RW is Read Write.
- uint32 is unsigned 32 bits integer.

Object	Attribute Name	RO/RW	Type	Description
Global	**version**	**RO**	**string**	Version of driver **eschgr**
LUN	**default_secs**	**RW**	**uint32**	Timeout for all commands not referenced below. Default: 30
	move_secs	**RW**	**uint32**	Timeout for the move command. Default: 1200
	readelem_secs	**RW**	**uint32**	Timeout for the read element status command. Default: 600
	initelem_secs	**RW**	**uint32**	Timeout for the initialize element status command. Default: 600

S

readaddr_secs	**RW**	**uint32**	Timeout for modesense 0x1D command. Default: 600
exchange_secs	**RW**	**uint32**	Timeout for the exchange command. Default: 600

I/O Load Balancing and Multi-Pathing Policies

The **eschgr** driver does not support load balancing and has minimal support for multi-pathing.

When the device is first opened after a system boot, a path is chosen and will remain fixed. If the path fails, the next open will pick a new path.

EXAMPLES

To display **scsimgr eschgr** plug-in general help and supported commands:

scsimgr -h -d eschgr

To get **eschgr** driver global statistics:

scsimgr get_stat -d eschgr

To clear **eschgr** driver global statistics:

scsimgr clear_stat -d eschgr

To get **eschgr** driver global status information:

scsimgr get_info -d eschgr

To display information about **eschgr** driver global attributes:

scsimgr get_attr -d eschgr

AUTHOR

SCSI class driver **eschgr** plug-in for **scsimgr** was developed by HP.

SEE ALSO

scsictl(1M), scsimgr(1M), autochanger(7), intro(7), scsi(7).

NAME

scsimgr_esdisk - SCSI class driver esdisk plug-in for scsimgr

DESCRIPTION

The SCSI class driver **esdisk** plug-in for **scsimgr** implements management and diagnostic operations specific to classes of devices bound to the **esdisk** driver. **esdisk** is the native HP-UX SCSI class driver that handles, by default, all block devices including the following types : direct access, CD/DVD, write-once read-multiple (WORM), and optical memory (OM).

The plug-in handles the following operations for driver **esdisk**:

- Displays and clears driver **esdisk** global statistics and the statistics it maintains on instances of LUNs bound to it, and on related LUN paths.
- Displays status and other information maintained by driver **esdisk** on LUNs bound to it.
- Gets, sets, and saves driver **esdisk** global, per-lun instance attributes or attributes for a set of devices bound to the driver.

Commands

The user can explicitly send the following **scsimgr** commands to driver **esdisk** plug-in by specifying the **-d esdisk** option:

clear_stat Clears statistics.

get_attr Displays information on attributes.

get_info Displays status and other information.

get_stat Displays statistics.

save_attr Saves value of attributes in a persistent store.

set_attr Set current values of attributes.

Note: Refer to *scsimgr*(1M) for the syntax of the above commands.

However, the only instance when it is necessary to explicitly send a command to the plug-in is when performing operations on objects global to driver **esdisk**: global statistics, attributes or status information. In all the other cases, **scsimgr** automatically invokes the plug-in to perform the driver specific part of the operation, when the operation applies to LUNs bound to driver **esdisk** or to their LUN paths.

Attributes

The following table lists driver **esdisk** specific attributes. Also, under the category "Device Set", it lists the attributes the **esdisk** driver can set at scopes; including, device type, vendor identifier, product identifier and product revision. On the concept of attribute and attribute scope, refer to *scsimgr*(1M).

Note: The following conventions are used :

- RO is Read Only.
- RW is Read Write.
- uint32 is unsigned 32 bit integer.
- Range of values for applicable attributes is listed.

Object	Attribute Name	RO/RW	Type	Description
Global	**version**	**RO**	**string**	Version of driver **esdisk**
LUN	**capacity**	**RO**	**uint32**	Device capacity in number of blocks
	block_size	**RO**	**uint32**	Block size in bytes
	path_fail_secs	**RW**	**uint32**	Delay in seconds before declaring a LUN path offline after failure of first I/O. Range: 0-600
	load_bal_policy	**RW**	**string**	I/O load balancing policy. May be: **round_robin**, **least_cmd_load**, **cl_round_robin**,

S

				preferred_path.
	infinite_retries _enable	**RW**	**boolean**	Enable or disable infinite retry of I/Os. May be: true: enable, false: disable.
	preferred_path	**RW**	**string**	Hardware path of the lunpath to use preferably for I/O transfer, when I/O load balancing policy is set to preferred_path.
Device Set	**transient_secs**	**RW**	**uint32**	Seconds to wait after a LUN transitioned out of **ONLINE** state before failling I/Os. Range: 0-600
	format_secs	**RW**	**uint32**	Timeout in secs of SCSI command FORMAT. Range: 0-0xFFFFFFFF
	start_unit_secs	**RW**	**uint32**	Timeout in secs of SCSI command START UNIT. Range: 0-0xFFFFFFFF
	max_retries	**RW**	**uint32**	Maximum number of I/O retries. Range: 1-0xFFFFFFFF
	path_fail_secs	**RW**	**uint32**	Timeout in secs before declaring a LUN path offline. Range: 0-600
	esd_secs	**RW**	**uint32**	Maximum time in secs for the transmission of an I/O. Range: 0-0xFFFFFFFF
	max_q_depth	**RW**	**uint32**	Maximum queue depth. Range: 1-0xFFFFFFFE
	load_bal_policy	**RW**	**string**	I/O load balancing policy. May be: **round_robin**, **least_cmd_load**, **cl_round_robin**, **preferred_path**.
	disable_flags	**RW**	**string**	A set of flags representing SCSI task management and other functions. If a flag is set, the corresponding function is disabled for the set of devices. The following flags are currently defined: **WCE**: Write Cache Enable, **RW16**: 16 bytes READ/WRITE CDB, **ABT**: SCSI task management function Abort Task Set, **CTS**: SCSI task management function Clear Task Set, **LR**: SCSI task management function LUN Reset, **WTR**: SCSI task management function Warm Target Reset, **CTR**: SCSI task management function Cold Target Reset, **BR**: Bus Reset, **PR**: Persistent Reservation, **WERO**: Persistent Reservation WERO (Write Exclusive Read-Only), **AERO**: Persistent Reservation AERO (Access Exclusive Read-Only).
	infinite_retries _enable	**RW**	**boolean**	Enable or disable infinite retries of I/Os. May be: true: enable, false: disable.

S

I/O Load Balancing Policy

The I/O load balancing policy attribute, **load_bal_policy**, is a tunable that controls how I/Os are distributed across the paths to a LUN:

- **round_robin**

  Paths are selected in a round robin manner. This is more appropriate when all the paths to the device have similar I/O turn-around characteristics.

- **least_cmd_load**

  The LUN path with the least number of active I/O requests is selected to execute the next I/O. This policy is appropriate when the paths to the LUN exhibit asymmetric latency characteristics. The load is distributed to optimize the bandwidth on each LUN path.

- **cl_round_robin** (cell aware round robin)

  This load balancing policy is applicable to HP cell-based platforms. The LUN paths are selected in a round robin manner within the locality of CPU on which the I/O was initiated, to ensure that memory access latencies are optimized.

- **preferred_path**

  The I/O path set in the preferred_path attribute is preferrably used for I/O transfer. If this I/O path is not available or if the preferred_path attribute was not set, any other path is selected for I/O transfer. This policy is useful for certain disk arrays, which may exhibit some performance degradation if I/Os are transferred via several I/O paths to a LUN simultaneously.

EXAMPLES

To display **scsimgr esdisk** plug-in general help and supported commands:

```
scsimgr -h -d esdisk
```

To get esdisk driver global statistics

```
scsimgr get_stat -d esdisk
```

To clear esdisk driver global statistics

```
scsimgr clear_stat -d esdisk
```

To get esdisk driver global status information

```
scsimgr get_info -d esdisk
```

To display information about esdisk driver global attributes

```
scsimgr get_attr -d esdisk
```

To set the load balancing policy for disk0 to preferred_path and set the I/O path to be used preferably

```
scsimgr set_attr -D /dev/rdisk/disk0  -a load_bal_policy=preferred_path
   -a preferred_path=0/3/1/0.0x21000020371972eb.0x0
```

To add a settable attribute scope corresponding to all disk devices from HP with product identifier "MSA VOLUME", for allowing modification of some settable attribute at this scope

```
scsimgr ddr_add -N "/escsi/esdisk/0x0/HP /MSA VOLUME  "
```

S

To persistently change the default I/O load balancing policy, I/O timeout, and maximum concurrent I/O for all disk devices from HP with product identifier "MSA VOLUME"

```
scsimgr save_attr -N "/escsi/esdisk/0x0/HP  /MSA VOLUME  "
        -a load_bal_policy=least_cmd_load -a esd_secs=60
        -a path_fail_secs=60
```

To disable write cache, Persistent Reservation and 16 bytes read/write CDB for all disk devices bound to the **esdisk** driver

```
scsimgr set_attr -N /escsi/esdisk -a disable_flags='WCE PR RW16'
```

AUTHOR

SCSI class driver **esdisk** plug-in for **scsimgr** was developed by Hewlett Packard Company.

SEE ALSO

diskinfo(1M), scsictl(1M), scsimgr(1M), intro(7), scsi(7), scsi_disk(7).

NAME

scsimgr_estape - SCSI class driver estape plug-in for scsimgr

DESCRIPTION

The SCSI class driver **estape** plug-in for **scsimgr** implements management and diagnostic operations specific to classes of devices bound to driver **estape**. **estape** is the native HP-UX SCSI class driver that handles, by default, all tape devices.

The plug-in handles the following operations for driver estape:

- Display and clear driver **estape** global statistics and the statistics it maintains on instances of LUNs bound to it, and on related LUN paths.
- Display status and other information maintained by driver **estape** on LUNs bound to it.
- Get, set and save driver **estape** global and per-lun instance attributes.

Commands

The user can explicitly send the following scsimgr commands to driver **estape** plug-in by specifying the **-d estape** option:

clear_stat	Clears statistics.
get_attr	Displays information on attributes.
get_info	Displays status and other information.
get_stat	Displays statistics.
save_attr	Saves value of attributes in a persistent store.
set_attr	Set current values of attributes.

Note: Refer to *scsimgr*(1M) for syntax of the above commands.

However, the only instances where it is necessary to explicitly send a command to the plug-in is when performing operations on objects global to driver **estape**: global statistics, attributes or status information. In all the other cases, **scsimgr** automatically invokes the plug-in to perform the driver specific part of the operation, when the operation applies to LUNs bound to driver estape or to their LUN paths.

Attributes

The following table lists driver estape specific attributes. For details on the concept of attribute refer to *scsimgr*(1M).

Note: The following conventions are used:

- RO is Read Only.
- RW is Read Write.
- VBM is Variable Block Mode.
- uint32 is unsigned 32 bits integer.
- uint64 is unsigned 64 bits integer.
- Range of values for applicable attributes is listed.

Object	Attribute Name	RO/RW	Type	Description
Global	**version**	**RO**	**string**	Version of driver **estape**.
	norewind_ close_disable	**RW**	**uint32**	Disables the ability to open a "rewind" device. Default: 0. Values: 0 (disabled), 1 (enabled).
	st_ats_enabled	**RW**	**uint32**	Determines whether to reserve the device on open and release on close. See *st_ats_enabled*(5). Default: 1.

S

				Values: 0 (disabled), 1 (enabled).
LUN	**default_secs**	**RW**	**uint32**	Timeout for all commands not referenced below. Default: 30.
	space_secs	**RW**	**uint32**	Timeout for the space command. Default: 1200.
	write_secs	**RW**	**uint32**	Timeout for the write command. Default: 600.
	read_secs	**RW**	**uint32**	Timeout for the read command. Default: 600.
	unload_secs	**RW**	**uint32**	Timeout for the unload command. Default: 600.
	rewind_secs	**RW**	**uint32**	Timeout for the rewind command. Default: 600.
	erase_secs	**RW**	**uint32**	Timeout for the erase command. Default: 18000.
	mt_type	**RW**	**uint32**	The type of device a particular LUN is associated with. Default: 0. Values: 0 = unknown, 5 = HPIB 9-track, 6 = DDS1, 7 = All other DDS/DAT, 8 = SCSI 9-track, 9 = QIC, 10 = 8mm, 11 = IBM 3480, STK 9XXX, STK T10000, 12 = Quantum DLT, 13 = Sony AIT, 14 = IBM 3590, 15 = LTO.
	default_blocksize	**RW**	**uint32**	Default blocksize. Default: 0. Values: 0 = variable, overridden by a custom DSF.
	default_ir	**RW**	**uint32**	Default immediate reporting. Default: 1. Values: 0 (disabled), 1 (enabled).
	close_marks	**RW**	**uint32**	Number of filemarks to indicate End of Data. Default: 2.
	num_partitions _supp	**RW**	**uint32**	Number of partitions supported Default: 1. Range: 1+
	characteristics	**RW**	**uint64**	Driver characteristics bitwise ORed together Default: 0. Values: 1 = Device supports setmarks, 2 = Logpage 31 contains capacity information, 4 = Logpage 38 contains capacity information, 8 = Device supports Reserve/Release.

S

best_density	**RW**	**uint32**	Tape density to write. Default: 0x7F. Values: 0xFFFFFFFF = Best density, 0x00 = Let the device choose the density, 0x7F = Do not modify tape density, other = a valid density code for the Mode Parameter Block Descriptor.
best_compression	**RW**	**uint32**	Compression Algorithm to use. Default: 0. Values: 0x00 = Compression Disabled, 0xDEF = default for drive, other = a valid compression value for the Data Compression Mode Page (0x0F).
clean_req_ sns_info	**RW**	**uint32**	The Key/Code/Qualifier representing "Cleaning Required" Default: 0xFFFFFFFF.

I/O Load Balancing and Multipathing Policies

The **estape** driver does not support load balancing and has minimal support for multipathing.

When the device is first opened after a system boot, a path is chosen and will remain fixed. If the path fails, the next open will pick a new path.

EXAMPLES

To display **scsimgr estape** plug-in general help and supported commands:

```
scsimgr -h -d estape
```

To get **estape** driver global statistics:

```
scsimgr get_stat -d estape
```

To clear **estape** driver global statistics:

```
scsimgr clear_stat -d estape
```

To get **estape** driver global status information:

```
scsimgr get_info -d estape
```

To display information about **estape** driver global attributes:

```
scsimgr get_attr -d estape
```

AUTHOR

SCSI class driver **estape** plug-in for **scsimgr** was developed by HP.

SEE ALSO

scsictl(1M), scsimgr(1M), st_ats_enabled(5), intro(7), scsi(7), scsi_tape(7).

NAME

sioc_io - SCSI pass-through interface

DESCRIPTION

SCSI devices are controlled by a device-specific driver, when one exists. Device-specific drivers, such as those for SCSI direct access (disk) and sequential access (tape) devices, coordinate device and driver states to accomplish correct logical device behavior. The **sioc_io** pass-through interface enables the use of SCSI devices and commands not normally supported by these device-specific drivers. It is composed of two ioctls: **SIOC_IO_EXT**, and **SIOC_IO**.

SIOC_IO_EXT is the pass-through interface introduced with HP-UX 11i V3 release. It is the recommended interface. It should be issued on persistent device files (see *intro*(7)). It allows to send the SCSI command through any of the available LUN paths or through a specific LUN path.

SIOC_IO is the pass-through interface that existed prior to HP-UX 11i V3. This interface is deprecated with HP-UX 11i V3 release. It is maintained for backward compatibility. It can be used on persistent device files or legacy device files. If issued on a persistent device file, the SCSI command is sent through any of the available LUN paths. If issued on a legacy device file, the SCSI command will be sent through any available LUN paths. However, if multi-pathing is disabled legacy device files (see **leg_mpath_enable** in *scsimgr*(1M)), the SCSI command will be sent only through the LUN path corresponding to the legacy device file.

All reserved fields in the data structure associated to the interface must be zero-filled.

The **SIOC_IO_EXT**/**SIOC_IO** ioctl allows an arbitrary SCSI command to be sent to a device. All details of the SCSI command protocol are handled automatically.

The data structure for the **SIOC_IO_EXT**/**SIOC_IO** ioctl is included from **<sys/scsi.h>**:

```
/* SCSI device control ioctls */

#define SIOC_IO_EXT         _IOWR('S', 102, esctl_io_t)
#define SIOC_IO             _IOWR('S', 22, struct sctl_io)

/* Structure for SIOC_IO_EXT ioctl */
typedef struct {
        int                     version;
        escsi_sctl_io_flags_t   flags;
        int                     max_msecs;
        uint32_t                cdb_length;
        uint32_t                data_length;
        ptr64_t                 data;
        union sense_data        sense;
        escsi_hw_path_t         lpt_hwp;
        uint32_t                data_xfer;
        uint32_t                sense_xfer;
        uint32_t                cdb_status;
        uint32_t                sense_status;
        uint8_t                 cdb[ESCSI_MAX_CDB_LEN];
        uint32_t                rsvd[32];  /* Reserved for
                                            * future use
                                            */
} esctl_io_t;

/* Structure for SIOC_IO ioctl */
typedef struct sctl_io {
        unsigned        flags;
        unsigned char   cdb_length;
        unsigned char   cdb[16];
        void            *data;
        unsigned        data_length;
        unsigned        max_msecs;
        unsigned        data_xfer;
        unsigned        cdb_status;
        unsigned char   sense[256];
```

S

```
        unsigned        sense_status;
        unsigned char   sense_xfer;
        unsigned char   reserved[64];
} sctl_io_t;
```

The following flags can be used to specify the **flags** field value of both **SIOC_IO_EXT** and **SIOC_IO** unless indicated otherwise:

SCTL_READ Data-in phase expected if the *data_length* field is non-zero. The absence of this flag implies that a data-out phase is expected if the *data_length* field is non-zero.

ESCTL_IO_LPT The SCSI command is to be issued on a given LUN path. This flag can only be specified with **SIOC_IO_EXT** ioctl. When specified the hardware path of the LUN path to use is specified in field *lpt_hwp*

The *cdb* field specifies the SCSI command bytes. The number of command bytes is specified by the *cdb_length* field. These command bytes are sent to the target device during the SCSI command phase.

The address of the data area for the data phase of the SCSI command is specified by the *data* field. The **data_length** field specifies the maximum number of data bytes to be transferred. A zero-valued *data_length* indicates that no data phase should occur. Most SCSI commands with a data phase expect the data length information to be included somewhere in the command bytes. The caller is responsible for correctly specifying both the *data_length* field and any *cdb* data length values. The length may not be larger than **SCSI_MAXPHYS** and some implementations further restrict this length.

The *max_msecs* field specifies the maximum time, in milliseconds, that the device should need to complete the command. If this period of time expires without command completion, the system might attempt recovery procedures to regain the device's attention. These recovery procedures might include abort tag, abort, and device and bus reset operations. A zero value in the *max_msec* field indicates that the timeout period is infinite and the system should wait indefinitely for command completion.

When the **SIOC_IO_EXT/SIOC_IO** ioctl call returns, all command processing has been completed. Most **SIOC_IO_EXT/SIOC_IO** ioctl calls will return zero (success). The resulting detailed ioctl data should be used to evaluate "success" or "failure" from the caller's perspective. The **cdb_status** field indicates the results of the **cdb** command. If the *cdb_status* field indicates a **S_CHECK_CONDITION** status, the *sense_status* field indicates the results of the SCSI **REQUEST SENSE** command used to collect the associated sense data. These status fields will contain one of the following values:

SCTL_INVALID_REQUEST The SCSI command request is invalid and thus not attempted.

SCTL_SELECT_TIMEOUT The target device does not answer to selection by the host SCSI interface (the device does not exist or does not respond).

SCTL_INCOMPLETE The device answered selection but the command is not completed (the device took too long or a communication failure occurred).

S_GOOD Device successfully completed the command.

S_CHECK_CONDITION Device indicated sense data is available.

S_CONDITION_MET Device successfully completed the command and the requested (search or pre-fetch) operation is satisfied.

S_BUSY Device indicated it is unable to accept the command because it is busy doing other operations.

S_INTERMEDIATE Device successfully completed this command, which is one in a series of linked commands (not supported, see *WARNINGS*).

S_I_CONDITION_MET Device indicated both **S_INTERMEDIATE** and **S_CONDITION_MET** (not supported, see *WARNINGS*).

S_RESV_CONFLICT Device indicated the command conflicted with an existing reservation.

S_COMMAND_TERMINATED Device indicated the command is terminated early by the host system.

S_QUEUE_FULL Device indicated it is unable to accept the command because its command queue is currently full.

The **data_xfer** field indicates the number of data bytes actually transferred during the data phase of the **cdb** command. This field is valid only when the *cdb_status* field contains one of the following values:

S_GOOD or **S_CHECK_CONDITION**. The *sense_xfer* field indicates the number of valid sense data bytes. This field is valid only when the *cdb_status* field contains the value **S_CHECK_CONDITION** and the *sense_status* field contains the value **S_GOOD**.

Security Restrictions

Use of the **SIOC_IO** ioctl requires the superuser or **DEVOPS** privilege, or device write permissions. See *privileges*(5) for more information about privileged access on systems that support fine-grained privileges.

EXAMPLES

Assume that *fildes* is a valid file descriptor for a persistent device file of a SCSI device, and *leg_fildes* is a valid file descriptor for a legacy device file of a SCSI device, and *lpt_hwp* contains a valid hardware path of a LUN path to the device. The first example attempts a SCSI **INQUIRY** command:

```
#include <sys/scsi.h>

esctl_io_t esctl_io;
#define MAX_LEN 255
unsigned char inquiry_data[MAX_LEN];

memset(&esctl_io, 0, sizeof(esctl_io)); /* clear reserved fields */
esctl_io.flags = SCTL_READ;   /* input data expected */
esctl_io.cdb[0] = CMDinquiry;
esctl_io.cdb[1] = 0x00;
esctl_io.cdb[2] = 0x00;
esctl_io.cdb[3] = 0x00;
esctl_io.cdb[4] = MAX_LEN;            /* allocation length */
esctl_io.cdb[5] = 0x00;
esctl_io.cdb_length = 6;              /* 6 byte command */
esctl_io.data = &inquiry_data[0];     /* data buffer location */
esctl_io.data_length = MAX_LEN;       /* maximum transfer length */
esctl_io.max_msecs = 10000;           /* allow 10 seconds for cmd */
if (ioctl(fildes, SIOC_IO_EXT, &esctl_io) < 0) {
        /* request is invalid */
} else {
        if ( esctl_io.cdb_status == S_GOOD) {
                /* success. display inquiry data */
        else {
                /* failure. process depending on cdb_status */
        }
}
```

The second example attempts a SCSI **INQUIRY** command via a specific LUN path.

S

```
#include <sys/scsi.h>

esctl_io_t esctl_io;
#define MAX_LEN 255
unsigned char inquiry_data[MAX_LEN];
memset(&esctl_io, 0, sizeof(esctl_io)); /* clear reserved fields */
esctl_io.flags = SCTL_READ | SCTL_IO_LPT;  /* input data
                                            * expected and command
                                            * to be sent on given
                                            * LUN path
                                            */
memcpy(&esctl_io.lpt_hwp, lpt_hwp, sizeof(lpt_hwp); /* specify
                                            * the hardware path of
                                            * LUN path through which
                                            * command must be sent
                                            */
esctl_io.cdb[0] = CMDinquiry;
esctl_io.cdb[1] = 0x00;
esctl_io.cdb[2] = 0x00;
esctl_io.cdb[3] = 0x00;
esctl_io.cdb[4] = MAX_LEN;            /* allocation length */
```

```
esctl_io.cdb[5] = 0x00;
esctl_io.cdb_length = 6;              /* 6 byte command */
esctl_io.data = &inquiry_data[0];     /* data buffer location */
esctl_io.data_length = MAX_LEN;       /* maximum transfer length */
esctl_io.max_msecs = 10000;           /* allow 10 seconds for cmd */
if (ioctl(fildes, SIOC_IO_EXT, &esctl_io) < 0) {
        /* request is invalid */
} else {
        if ( esctl_io.cdb_status == S_GOOD) {
                /* success. display inquiry data */
        else {
                /* failure. process depending on cdb_status */
        }
}
```

The following example attempts a SCSI **TEST UNIT READY** command and checks to see if the device is ready, not ready, or in some other state.

```
#include <sys/scsi.h>

struct sctl_io sctl_io;

memset(&sctl_io, 0, sizeof(sctl_io)); /* clear reserved fields */
sctl_io.flags = 0;                   /* no data transfer expected */
sctl_io.cdb[0] = 0x00;               /* can use CMDtest_unit_ready */
sctl_io.cdb[1] = 0x00;
sctl_io.cdb[2] = 0x00;
sctl_io.cdb[3] = 0x00;
sctl_io.cdb[4] = 0x00;
sctl_io.cdb[5] = 0x00;
sctl_io.cdb_length = 6;              /* 6 byte command */
sctl_io.data = NULL;                 /* no data buffer is provided */
sctl_io.data_length = 0;             /* do not transfer data */
sctl_io.max_msecs = 10000;           /* allow 10 seconds for cmd */
if (ioctl(leg_fildes, SIOC_IO, &sctl_io) < 0) {
        /* request is invalid */
}
else if (sctl_io.cdb_status == S_GOOD) {
        /* device is ready */
}
else if (sctl_io.cdb_status == S_BUSY ||
         (sctl_io.cdb_status == S_CHECK_CONDITION &&
          sctl_io.sense_status == S_GOOD &&
          sctl_io.sense_xfer > 2 &&
          (sctl_io.sense[2] & 0x0F) == 2)) {
         /* can use sense_data */
        /* device is not ready */
} else {
        /* unknown state */
}
```

S

WARNINGS

Incorrect use of **sioc_io** operations (even those attempting access to non-existent devices) can cause data loss, system panics, and device damage.

The **SIOC_EXCLUSIVE** ioctl should be used to gain exclusive access to a device prior to attempting **SIOC_IO** commands. If exclusive access is not obtained, **SIOC_IO** commands will be intermixed with device-specific driver commands, which can lead to undesirable results.

Device-specific drivers can reject inappropriate or troublesome **SIOC_IO** commands. However, since not all such operations are known and detected, care should be exercised to avoid disrupting device-specific drivers when using commands that modify internal device states.

Most SCSI commands have a logical unit number (LUN) field. Parallel SCSI implementations on the HP-UX operating system select logical units via the SCSI **IDENTIFY** message. The LUN portion of the **cdb** should normally be set to zero, even when the LUN being accessed is not zero.

Use of linked commands is not supported.

Most SCSI commands with a data phase expect the data length information to be included somewhere in the command bytes. Both the *data_length* field and any *cdb* data length values must be correctly specified to get correct command results.

Very large (or infinite) timeout values can cause a parallel SCSI bus (potentially the entire system) to hang.

Device and/or bus reset operations can be used to regain a device's attention when a timeout expires.

Resetting a device can cause I/O errors and/or loss of cached data. This can result in loss of data and/or system panics.

Obtaining SCSI **INQUIRY** data by use of the **SIOC_INQUIRY** ioctl instead of by use of the **SIOC_IO** ioctl is generally preferable since SCSI implementations on the HP-UX operating system synchronize access of inquiry data during driver open calls.

Since communication parameters can be affected by device-specific driver capabilities, device-specific driver use might result in communication parameter changes.

FILES

/usr/include/sys/scsi.h
/usr/include/sys/scsi_ctl.h

SEE ALSO

ioctl(2), privileges(5), intro(7), scsi(7), scsi_ctl(7).

NAME

slp_syntax - SLP Service Type Syntax

DESCRIPTION

The SLP API expects service type information to be passed while querying for SLP service information and also while registering and deregistering services. The SLP API accepts service type information in URL format also.

The service type string contains the following information.

Name of the service type.

Naming Authority responsible for the service name.

The service type string is of the form:

service **:** *abstract-type.naming-authority* **:** *concrete-type*

The *abstract-type* is a short descriptive string that describes the type of service.

The *naming-authority* is the name of the organization that named the service. The *naming-authority* is optional, but if it is omitted, then IANA is assumed to be the naming authority and IANA requires service-types to be registered (see RFC 2609).

concrete-type, also optional, is a kind of sub-type of the abstract-type.

For example,

printer is an abstract type (owned by IANA) and **printer:lpr** is a concrete type (owned by IANA).

The official definition of Service Type strings can be found in RFC 2609, "Service Templates and Service Schemes".

Examples of Service Type Strings

weather.nasa:wtp	A (fictitious) weather service type owned by NASA that uses WTP protocol.
weather.nasa:swtp	A (fictitious) weather service type owned by NASA that uses SWTP protocol.
chat.superchat	A chat service type owned by SuperChat.
printer.samba	A samba printer service type.
ftp	An IANA ftp service type.
telnet	An IANA telnet service type.

Comparing Service Types

Since service types are important in determining the URLs that are returned by the **SLPFindSrvs()** call, you should understand how services are compared. Suppose that three services were registered with **SLPReg()** using a *srvtype* of **printer:lpr**, **printer** and **printer.acme**. If a client program calls **SLPFindSrvs()** with a *srvtype* of **service:printer**, the urls for both **printer:lpr** and **printer** are returned (**printer.acme** is not). However, if **SLPFindSrvs()** is called with *srvtype* of **printer:lpr** or **printer.acme**, then the urls for **printer:lpr** or **printer.acme** would be returned. In other words, if a concrete-type is used, only services with the same abstract and concrete-type are returned. If only the abstract type is used, then all services of that abstract type (and naming authority) are returned.

S

SLP Service URL Syntax

SLP APIs accept service type strings in URL syntax format. URL strings are passed as parameters to **SLPReg()**, **SLPDeReg()**, **SLPFindSrvs()**, and **SLPParseSrvURL()** functions and returned as a result to the **SLPSrvURLCallback()** callback function. SLP defines a special type of URL called a Service URL that MUST be used when calling SLP API functions. The syntax of a service URL is:

SLP Service URL = service: *service-type* **://** *addrspec*

service-type is a service type as explained above. *addrspec* can be any address that fits URL syntax and can be translated as a network location. The **service:** and **://** strings are required.

Service URL Examples

```
service:weather.nasa:wtp://weather.nasa.com:12000
service:weather.nasa:swtp://weather.nasa.com:12001
service:chat.superchat://chat.superchat.com;auth=ldap
```

SLP requires you to use Service URLs. API functions will return **SLP_PARSE_ERROR** if you do not. Service URLs are required because the SLP API designers do not allow the service-type to be passed in as a parameter to the **SLPDeReg()** call. Without the service-type, **SLPDeReg()** does not allow the caller to distinguish between services of varying types that were registered with the same standard URL.

The **SLPFindSrvs()** function expects the search strings to be passed in LDAPv3 Search Filter Syntax.

SEE ALSO

slpd(1M), libslp(3N), slp.reg(4).

NAME

socket - interprocess communications

DESCRIPTION

Sockets are communication endpoints that allow processes to communicate either locally or remotely. They are accessed by means of a set of system calls (see *socket*(2)).

The following **ioctl()** requests are defined in <**sys/ioctl.h**> (see *ioctl*(2)):

FIOSNBIO If the int with the address *arg* is non-zero, the socket is put into non-blocking mode. Otherwise, the socket is put into blocking mode. Blocking mode is the default. The **FIONBIO** request is equivalent to the **FIOSNBIO** request, although using **FIONBIO** is not recommended. See *accept*(2), *connect*(2), *recv*(2), and *send*(2) for an explanation of how non-blocking mode is used.

FIONREAD For SOCK_STREAM sockets, the number of bytes currently readable from this socket is returned in the integer with the address *arg*. For SOCK_DGRAM sockets, the number of bytes currently readable, plus the size of the *sockaddr* structure (defined in <**sys/socket.h**>), is returned in the integer with the address *arg*.

SIOCATMARK For SOCK_STREAM TCP sockets, on return the integer with the address *arg* is non-zero if the inbound TCP stream has been read up to where the out-of-band data byte starts. Otherwise, the inbound TCP stream has not yet been read up to where the out-of-band data byte starts. For sockets other than SOCK_STREAM TCP sockets, on return the integer with the address *arg* is always zero.

SIOCSPGRP This request sets the process group or process ID associated with the socket to be the value of the integer with the address *arg*. A process group or process ID associated with the socket in this manner is signaled when the state of the socket changes: **SIGURG** is delivered upon the receipt of out-of-band data; **SIGIO** is delivered if the socket is asynchronous, as described in **FIOASYNC** below. If the value of the integer with the address *arg* is positive, the signal is sent to the process whose process ID matches the value specified. If the value is negative, the signal is sent to all the processes that have a process group equal to the absolute value of the value specified. If the value is zero, no signal is sent to any process. It is necessary to issue this request with a non-zero integer value to enable the signal delivery mechanism described above. The default for the process group or process ID value is zero.

SIOCGPGRP This request returns the process group or process ID associated with the socket in the integer with the address *arg*. See the explanation for **SIOCSPGRP** above for more details on the meaning of the integer value returned.

FIOASYNC If the integer whose address is *arg* is non-zero, this request sets the state of the socket as asynchronous. Otherwise, the socket is put into synchronous mode (the default). Asynchronous mode enables the delivery of the **SIGIO** signal when either of the following conditions is met.

- New data arrives.
- For connection-oriented protocols, whenever additional outgoing buffer space becomes available or the connection is established or broken.

The process group or process ID associated with the socket must be non-zero in order for **SIGIO** signals to be sent. The signal is delivered according to the semantics of **SIOCSPGRP** described above.

The *fcntl*(2) **O_NDELAY** and **O_NONBLOCK** flags (defined in <**fcntl.h**>) are supported by sockets. If the **O_NONBLOCK** flag is set, the socket is put into POSIX-style non-blocking mode. If the **O_NDELAY** flag is set, the socket is put into non-blocking mode. Otherwise, the socket is put into blocking mode. Blocking mode is the default. See *accept*(2), *connect*(2), *recv*(2), and *send*(2) for an explanation of how these forms of non-blocking mode are used.

Since the **fcntl() O_NONBLOCK** and **O_NDELAY** flags and **ioctl() FIOSNBIO** requests are supported, the following clarifies on how these features interact. If the **O_NONBLOCK** or **O_NDELAY** flag has been set, **recv()** and **send()** requests behave accordingly, regardless of any **FIOSNBIO** requests. If neither the **O_NONBLOCK** flag nor the **O_NDELAY** flag has been set, **FIOSNBIO** requests control the the behavior of **recv()** and **send()**.

DEPENDENCIES

AF_CCITT Only

Only the **FIOSNBIO**, **FIONREAD**, **SIOCGPGRP**, and **SIOCSPGRP ioctl()** requests are defined for **af_ccitt** sockets.

AUTHOR

socket was developed by the University of California, Berkeley.

SEE ALSO

fcntl(2), getsockopt(2), ioctl(2), socket(2).

NAME

streamio - STREAMS ioctl commands

SYNOPSIS

#include <sys/types.h>
#include <stropts.h>

int ioctl(int *fildes*, **int** *command*, **... /*** *arg* ***/);**

DESCRIPTION

STREAMS **ioctl** commands are a subset of the **ioctl()** system calls which perform a variety of control functions on streams.

fildes is an open file descriptor that refers to a stream. *command* determines the control function to be performed as described below. *arg* represents additional information that is needed by this command. The type of *arg* depends upon the command, but it is generally an integer or a pointer to a *command*-specific data structure. The *command* and *arg* are interpreted by the stream head. Certain combinations of these arguments may be passed to a module or driver in the stream.

Since these STREAMS commands are a subset of **ioctl**, they are subject to the errors described there. In addition to those errors, the call will fail with **errno** set to [EINVAL], without processing a control function, if the stream referenced by *fildes* is linked below a multiplexor, or if *command* is not a valid value for a stream.

Also, as described in **ioctl**, STREAMS modules and drivers can detect errors. In this case, the module or driver sends an error message to the stream head containing an error value. This causes subsequent system calls to fail with **errno** set to this value.

The following **ioctl** commands, with error values indicated, are applicable to all STREAMS files:

I_ATMARK Allows the user to see if the current message on the stream head read queue is "marked" by some module downstream. *arg* determines how the checking is done when there are multiple marked messages on the stream head read queue. It may take the following values:

- **ANYMARK** Checks if the message is marked.
- **LASTMARK** Checks if the message is the last one that is marked on the queue.

If both **ANYMARK** and **LASTMARK** are set, **ANYMARK** supersedes **LASTMARK**.

The return value is 1 if the mark condition is satisfied and 0 otherwise.

I_CANPUT Checks if a certain band is writable. *arg* is set to the priority band in question. The return value is 0 if the priority band *arg* is flow controlled, 1 if the band is writable, or –1 on error.

I_CKBAND Check if the message of a given priority band exists on the stream head read queue. This returns 1 if a message of a given priority exists, or –1 on error. *arg* should be an integer containing the value of the priority band in question.

I_FDINSERT Creates a message from user specified buffer(s), adds information about another stream and sends the message downstream. The message contains a control part and an optional data part. The data and control parts to be sent are distinguished by placement in separate buffers, as described below.

arg points to a **strfdinsert** structure which contains the following members:

```
struct strbuf      ctlbuf;
struct strbuf      databuf;
long               flags;
int                fildes;
int                offset;
```

The *len* field in the **ctlbuf strbuf** structure (see *putmsg*(2)) must be set to the size of a pointer plus the number of bytes of control information to be sent with the message. *fildes* in the **strfdinsert** structure specifies the file descriptor of the other stream. *offset*, which must be word-aligned, specifies the number of bytes beyond the beginning of the control buffer where **I_FDINSERT** will store a pointer. This pointer will be the address of the read queue structure of the driver for the

S

streams corresponding to *fildes* in the **strfdinsert** structure. The *len* field in the **databuf strbuf** structure must be set to the number of bytes of data information to be sent with the message or zero if no data part is to be sent.

flags specifies the type of message to be created. An ordinary (non-priority) message is created if *flags* is set to 0, a high priority message is created if *flags* is set to **RS_HIPRI**. For normal messages, **I_FDINSERT** will block if the stream write queue is full due to internal flow control conditions. For high priority messages, **I_FDINSERT** does not block on this condition. For normal messages, **I_FDINSERT** does not block when the write queue is full and the **O_NONBLOCK** is set. Instead, it fails and sets **errno** to [EAGAIN].

I_FDINSERT also blocks, unless prevented by the lack of internal resources, waiting for the availability of message blocks, regardless of priority or whether **O_NONBLOCK** has been specified. No partial message is sent.

I_FDINSERT can also fail if an error message was received by the stream head of the stream corresponding to *fildes* in the **strfdinsert** structure. In this case, **errno** will be set to the value in the message.

I_FIND Compares the names of all modules currently present on the stream to the name specified in *arg*. The command returns a value of 1 if the module is present and a value of 0 (zero) if the module is not present.

I_FLUSH This request flushes all input and/or output queues, depending on the value of *arg*. Valid *arg* values are:

FLUSHRW Flush write and read queues.

FLUSHW Flush write queues.

FLUSHR Flush read queues.

If a pipe or FIFO does not have any modules pushed, the read queue of the streams head on either end is flushed depending on the value of *arg*.

If **FLUSHR** is set and *fildes* is a pipe, the read queue for that end of the pipe is flushed and the write queue for the other end is flushed. If *fildes* is a FIFO, both queues are flushed.

If **FLUSHW** is set and *fildes* is a pipe and the other end of the pipe exists, the read queue for the other end of the pipe is flushed and the write queue for this end is flushed. If *fildes* is a FIFO, both queues of the FIFO are flushed.

If **FLUSHRW** is set, all read queues are flushed, that is the read queue for the FIFO and the read queue of boths ends of the pipe are flushed.

Correct flushing handling of a pipe or FIFO with modules pushed is achieved via the **pipemod** module. This module should be the first module pushed onto a pipe so that it is at the midpoint of the pipe itself.

I_FLUSHBAND

Flushes a particular band of messages. *arg* points to a **bandinfo** structure that has the following members:

```
unsigned char      bi_pri:
int                bi_flag;
```

The value of the *bi_flag* field can be **FLUSHR**, **FLUSHW**, or **FLUSHRW** as described for the **I_FLUSH** command.

I_GETBAND Returns the priority band of the first message on the stream head read queue in the integer referenced by *arg*.

I_GETCLTIME

Returns the close time delay in the long pointed by *arg*.

I_SETCLTIME

Allows the user to set the time that the stream head will delay when a stream is closing, and there is data on the write queues. Before closing each module and driver, the stream head will delay for the specified amount of time to allow the data to drain. If, after the delay, data is still present, data will be flushed. *arg* is a pointer to the number of milliseconds to delay, rounded up to the nearest valid value on the system. The default is fifteen seconds.

I_GETSIG Returns the events for which the calling process has registered to receive a **SIGPOLL** signal. Events are returned as in *arg* bitmask as defined for the **I_SETSIG** command.

I_GRDOPT Returns the current read mode setting in an *int* pointed to by the argument *arg*. Read modes are described in *read*(2).

I_GWROPT Returns the current write mode setting, as described in **I_SWROPT**, in the *int* that is pointed to by the argument *arg*.

I_LINK Connects two streams, where *fildes* is the file descriptor of the stream connected to the multiplexing driver, and *arg* is the file descriptor of the stream connected to another driver. The stream designated by *arg* gets connected below the multiplexing driver. **I_LINK** requires the multiplexing driver to send an acknowledgement message to the stream head regarding the linking operation. This call returns a multiplexor ID number (an identifier used to disconnect the multiplexor, see **I_UNLINK**) on success, and –1 on failure.

I_LIST Allows the user to list all the module names on the stream, up to and including the topmost driver name. If *arg* is **NULL**, the return value is the number of modules, including the driver, that are on the stream pointed to by *fildes*. This allows the user to allocate enough space for the module names. If *arg* is not **NULL**, it should point to a **str_list** structure that has the following members:

```
int       sl_nmods;
struct    str_mlist   *sl_modlist;
```

The **str_mlist** structure has the following member:

```
char    l_name[FMNAMESZ+1];
```

sl_nmods indicates the number of entries the user has allocated in the array. On success, the return value is 0, **sl_modlist** contains the list of module names, and **sl_nmods** indicates the number of entries that have been filled in.

I_LOOK Retrieves the name of the module located just below the streams head of the stream pointed to by *fildes*, and places it in a null terminated character string pointed at by *arg*. The buffer pointed to by *arg* should be at least **FNAMESZ+1** bytes long. A **#include <stropts.h>** declaration is required.

I_NREAD Counts the number of data bytes in data blocks in the first message on the stream head read queue, and places this value in the location pointed to by *arg*. The return value for the command is the number of messages on the stream head read queue. For example, if zero is returned in *arg*, but the **ioctl** return value is greater than zero, this indicates that a zero-length message is next on the queue.

I_PEEK Allows the user process to look (peek) at the contents of the first message on the stream head read queue. This is done without taking the message off the queue. The **I_PEEK ioctl** operates the same way as the **getmsg()** function, except that it does not remove the message. The *arg* parameter points to a **strpeek** structure (in the **<stropts.h>** header file) with the following members:

```
struct strbuf     ctlbuf;
struct strbuf     databuf;
long              flags;
```

The **strbuf** structure pointed to by **ctlbuf** and **databuf** has the following members:

```
int    maxlen;
int    len;
char   *buf
```

The *maxlen* field of the **strbuf** structure must specify the number of bytes of control or data information to be retrieved. The *flags* field can be set to **RS_HIPRI** or 0 (zero). If this field is set to **RS_HIPRI**, the **I_PEEK ioctl** looks for a high priority message on the queue. If the field is set to 0, the **I_PEEK ioctl** looks at the

S

first message on the queue.

The **I_PEEK** returns a 1 if a message was retrieved, and returns a value of 0 (zero) if no message was found; it does not wait for a message. Upon successful completion, **ctlbuf** specifies control information in the control buffer, **databuf** specifies data information in the data buffer, and *flags* contains **RS_HIPRI** or 0 (zero).

I_PLINK Connects two streams, where *fildes* is the file descriptor of the stream connected to the multiplexing driver, and *arg* is the file descriptor of the stream connected to another driver. The stream designated by *arg* gets connected via a persistent link below the multiplexing driver. **I_PLINK** requires the multiplexing driver to send an acknowledgement message to the stream head regarding the linking operation. This call creates a persistent link which can exist even if the file descriptor associated with the upper stream to the multiplexing driver is closed. This call returns a multiplexor ID number (an identifier that may be used to disconnect the multiplexor, see **I_PUNLINK**) on success and –1 on failure.

The **I_PLINK ioctl** can also fail if it is waiting for the multiplexing driver to acknowledge the link request and an error (**M_ERROR**) message, or hangup (**M_HANGUP**) message is received at the stream head for *fildes*. In addition, an error can be returned in an **M_IOACK** or **M_IONAK** message. When these occur, the **I_PLINK** fails with **errno** set to the value in the message.

I_POP Removes the module just below the stream head of the stream pointed to by *fildes*. To remove a module from a pipe requires that the module was pushed on the side it is being removed from. *arg* should be 0 in an **I_POP** request.

I_PUNLINK Disconnects the two streams specified by *fildes* and *arg* that are connected with a persistent link. *fildes* is the file descriptor of the stream connected to the multiplexing driver. *arg* is the multiplexor ID number that was returned by **I_PLINK** when a stream was linked below the multiplexing driver. If *arg* is **MUXID_ALL**, then all streams which are persistent links to *fildes* are disconnected. As in **I_PLINK**, this command requires the multiplexing driver to acknowledge the unlink.

I_PUSH Pushes the module whose name is pointed by *arg* onto the top of the current stream, just below the stream head. If the stream is a pipe, the module will be inserted between the streams heads of both ends of the pipe. It then calls the open routine of the newly-pushed module.

I_RECVFD Retrieves the file descriptor associated with the message sent by an **I_SENDFD** **ioctl** over a stream pipe. *arg* is a pointer to a data buffer large enough to hold a **strrecvfd** data structure containing the following members:

```
int       fd;
uid_t     uid;
gid_t     gid;
char      fill[8]
```

fd is an integer file descriptor. **uid** and **gid** are the user ID and group ID, respectively, of the sending stream.

If **O_NONBLOCK** is clear, **I_RECVFD** will block until a message is present at the stream head. If **O_NONBLOCK** is set, **I_RECVFD** will fail with **errno** set to [EAGAIN] if no message is present at the stream head.

If the message at the stream head is a message sent by a **I_SENDFD**, a new user file descriptor is allocated for the file pointer contained in the message. The new file descriptor is placed in the *fd* field of the **strrecvfd** structure. The structure is copied into the user data buffer pointed to by *arg*.

I_SENDFD Requests the stream associated with *fildes* to send a message, containing a file pointer, to the stream head at the other end of a stream pipe. The file pointer corresponds to *arg*, which must be an open file descriptor.

I_SENDFD converts *arg* into the corresponding system file pointer. It allocates a message block and inserts the file pointer in the block. The user ID and group ID associated with the sending process are also inserted. This message is placed directly on the read queue of the stream head at the other end of the stream pipe to which it is connected.

I_SETCLTIME
Lets the user process set the time that the stream head delays when the stream is closing and the write queues contain data. The *arg* parameter contains a pointer to the number of milliseconds to delay, rounded up to the nearest legal value on the system. The default time is 15 seconds.

Before STREAMS modules and drivers are closed, the stream head delays for the specified amount of time. This allows the data on the write queues to drain. If data is still present on the writes queues after the delay, the queues are flushed.

I_SETSIG Informs the stream head that the user wants the kernel to issue the **SIGPOLL** signal (see *signal*(2)) when a particular event has occurred on the stream associated with *fildes*. **I_SETSIG** supports an asynchronous processing capability in STREAMS. The value of *arg* is a bitmask that specifies the events for which the user should be signaled. It is the bitwise-OR of any combination, except where noted, of the following constants:

S_BANDURG When used in conjunction with **S_RDBAND**, **SIGURG** is generated instead of **SIGPOLL** when a priority message reaches the front of the stream head read queue.

S_ERROR An **M_ERROR** message has reached the stream head.

S_HANGUP An **M_HANGUP** message has reached the stream head.

S_HIPRI A high priority message is present on the stream head read queue. This is set even if the message is of zero length.

S_INPUT Any message other than an **M_PCPROTO** has arrived on a stream head read queue. This event is maintained for compatibility with prior releases. This is set even if the message is of zero length.

S_MSG A STREAMS signal message that contains the **SIGPOLL** signal has reached the front of the stream head read queue.

S_OUTPUT The write queue just below the stream head is no longer full. This notifies the user that there is room on the queue for sending (or writing) data downstream.

S_RDBAND A priority band message (band > 0) has arrived on a stream head read queue. This is set even if the message is of zero-length.

S_RDNORM An ordinary (non-priority) message has arrived on a stream head read queue. This is set even if the message is of zero-length.

S_WRBAND A priority band greater than 0 of a queue downstream exists and is writable. This notifies the user that there is room on the queue for sending (or writing) priority data downstream.

S_WRNORM This event is the same as **S_OUTPUT**.

A user process may choose to be signaled only of high priority messages by setting *arg* bitmask to the value **S_HIPRI**.

Processes that want to receive **SIGPOLL** signals must explicitly register to receive them using **I_SETSIG**. If several processes register to receive the signal for the same event on the same stream, each process will be signaled when the event occurs.

If the value of *arg* is zero, the calling process will be unregistered and will not receive further **SIGPOLL** signals.

I_SRDOPT Sets the read mode (see *read*(2)) using the value of the argument *arg*. Valid *arg* values are:

RNORM Byte-stream mode (default).

RMSGD Message-discard mode.

RMSGN Message-nondiscard mode.

Setting both **RMSGD** and **RMSGN** is an error. **RMSGD** and **RMSGN** override **NORM**.

In addition, treatment of control messages by the stream head may be changed by setting the following flags in *arg*:

RPROTNORM Fail **read** with EBADMSG if a control message is at the front of the stream head read queue. This is the default behavior.

RPROTDAT Deliver the control portion of a message as data when a user issues **read**.

RPROTDIS Discard the control portion of a message, delivering any data portion, when a user issues a **read**.

I_STR Constructs an internal STREAMS **ioctl** message from the data pointed to by *arg*, and sends that message downstream.

This mechanism is provided to send user **ioctl** requests to downstream modules and drivers. It allows information to be sent with the **ioctl**, and will return to the user any information sent upstream by the downstream recipient. **I_STR** blocks until the system responds with either a positive or negative acknowledgement message, or until the request "times out" after some period of time. If the request times out, it fails with **errno** set to ETIME.

At most, one **I_STR** can be active on a stream. Further **I_STR** calls will block until the active **I_STR** completes at the stream head. The default timeout intervals for these requests is 15 seconds. The **O_NONBLOCK** (see *open*(2)) flags have no effect on this call.

To send requests downstream, *arg* must point to a **strioctl** structure which contains the following members:

```
int     ic_cmd;
int     ic_timout;
int     ic_len;
char    *ic_dp;
```

ic_cmd is the internal **ioctl** command intended for the downstream module or driver and **ic_timout** is the number of seconds (–1 =infinite, 0 = use default, >0 = as specified) an **I_STR** request will wait for acknowledgement before timing out. The default timeout is infinite. **ic_len** is the number of bytes in the data argument and **ic_dp** is a pointer to the data argument. The **ic_len** field has two uses: on input, it contains the length of the data argument passed in, and on return from the command, it contains the number of bytes being returned to the user (the buffer pointed to by **ic_dp** should be large enough to contain the maximum amount of data that any module or driver in the stream can return). The stream head will convert the information pointed to by **strioctl** structure to an internal **ioctl** command message and send it downstream.

I_SWROPT Sets the write mode using the value of the argument *arg*. Legal bit settings for *arg* are:

SNDZERO Sends a zero-length message downstream when a write of 0 bytes occurs. To not send a zero-length message when a write of 0 bytes occurs, this bit must not be set in *arg*.

I_UNLINK Disconnects the two streams specified by *fildes* and *arg*. *fildes* is the file descriptor of the stream connected to the multiplexing driver. *arg* is the multiplexor ID number that was returned by the **I_LINK**. If *arg* is **MUXID_ALL**, then all streams which were linked to *fildes* are disconnected. As in **I_LINK**, this command requires the multiplexing driver to acknowledge the unlink.

RETURN VALUE

Unless specified differently for a command, the return value for a STREAMS **ioctl()** call is 0 (zero) on success and –1 (minus one) on failure.

ERRORS

A STREAMS **ioctl** command fails without performing the function and with **errno** set to [EINVAL] if:

- The stream referred to by *fildes* is linked below a multiplexing driver.

- The *command* parameter is not a valid value for the stream.

In addition, if any of the following conditions occur, the STREAMS **ioctl** commands return the corresponding value:

I_ATMARK

[EINVAL] *arg* has an illegal value.

I_CANPUT

[EINVAL] *arg* has an illegal value.

I_CKBAND

[EINVAL] *arg* has an illegal value.

I_FDINSERT

[EINVAL] The *fildes* parameter in the **strfdinsert** structure is an invalid open file descriptor.

[EINVAL] The size of the pointer plus *offset* exceeds the value of the *len* field for the buffer specified through *ctlptr*.

[EINVAL] *offset* does not specify a properly aligned location in the data buffer.

[EINVAL] *flags* contains an undefined value.

[EFAULT] *arg* points, or **ctrlbuf** or **databuf** is outside the allocated address space.

[EAGAIN] The **ioctl** request failed because a non-priority message was to be created, the **O_NONBLOCK** option was set, and the stream's write queue was full because of internal flow control conditions.

[ENOSR] Buffers could not be allocated for the message that was to be created due to insufficient STREAMS memory resources.

[ENXIO] A hangup was received on the stream specified by *fildes* in the **I_FDINSERT ioctl** call or on the stream specified by *fildes* in the **strfdinsert**.

[ERANGE] The value of the *len* field for the buffer specified through **databuf** does not fall within the range for the minimum and maximum sizes of packets for the top-most module on the stream.

[ERANGE] The value of the *len* field for the buffer specified through **databuf** is larger than the maximum allowable size for the data part of a message.

[ERANGE] The value of the *len* field for the buffer specified through **ctlbuf** is larger than the maximum allowable size for the control part of a message.

The **I_FDINSERT ioctl** can also fail if an error (**M_ERROR**) message was received by the stream specified by the *fildes* field in the **strfdinsert** structure. In this case, **errno** is set to the error value in the error message.

S

I_FIND

[EINVAL] *arg* does not contain a valid module name.

[EFAULT] *arg* points outside the allocated address space.

I_FLUSH

[ENOSR] Could not allocate buffers for flush operation because of a lack of STREAMS memory resources.

[EINVAL] The *arg* parameter is an invalid value.

[ENXIO] A hangup was received on *fildes*.

I_FLUSHBAND

[EINVAL] The *bi_pri* parameter value exceeds the maximum band, or the *bi_flag* parameter is not **FLUSHR**, **FLUSHW**, or **FLUSHRW**.

I_GETBAND

[ENODATA] No message exists on the stream head read queue.

I_GETSIG

[EINVAL] User process is not registered to receive the **SIGPOLL** signal.

[EFAULT] *arg* points outside the allocated address space.

I_GRDOPT

[EFAULT] *arg* is pointing outside the allocated address space.

I_LINK

[EAGAIN] Temporarily unable to allocate storage to perform the linking operation.

[EBADF] The *arg* parameter not a valid open file descriptor.

[ENXIO] A hangup was received on *fildes*.

[EINVAL] The stream referred to by *fildes* does not support multiplexing.

[EINVAL] The file referred to by *arg* is not a stream, or the stream is already linked under a multiplexor.

[EINVAL] The link operation would cause a "cycle" in the resulting multiplexing configuration. In other words, the driver referred to by the *arg* parameter is linked into this configuration at multiple places

[ENOSR] Not enough STREAMS memory resources to allocate storage for this command.

[ETIME] Acknowledgement message not received at stream head before timeout.

The **I_LINK ioctl** can also fail if an **M_ERROR** or **M_HANGUP** message is received at the stream head for *fildes* before receiving the driver acknowledgement. In addition, an error can be returned in an **M_IOCACK** or **M_IOCNAK** message. When these occur, the **I_LINK ioctl** fails with **errno** set to the value in the message.

I_LIST

[EINVAL] **sl_nmods** is less than 1.

[EAGAIN] Could not allocate buffers.

I_LOOK

[EINVAL] There are no modules in the stream.

[EFAULT] *arg* points outside the allocated address space.

I_NREAD

[EFAULT] *arg* is pointing outside the allocated address space.

S

I_PEEK

[EINVAL] The *flags* parameter is an illegal value.

[EFAULT] *arg* points, or **ctrlbuf** or **databuf** is, outside the allocated address space.

[EBADMSG] Message to be looked at is not valid for the **I_PEEK** command.

I_PLINK

[ENXIO] A hangup was received on the stream referred to by the *fildes* parameter.

[ETIME] A timeout occurred before an acknowledgement message was received at the stream head.

[EAGAIN] Temporarily unable to allocate storage to perform the linking operation.

[EBADF] *arg* is not a valid open file descriptor.

[EINVAL] The stream referred to by *fildes* does not support multiplexing.

[EINVAL] The file referred to by *arg* is not a stream or is already linked under a multiplexing driver.

[EINVAL] The link operation would cause a "cycle" in the resulting multiplexing configuration. In other words, the driver referred to by *arg* is linked into the configuration at multiple places.

I_POP

[EINVAL] There are not modules in the stream.

[ENXIO] Error value returned by the module being popped.

[ENXIO] A hangup was received on *fildes*.

I_PUNLINK

[ENXIO] A hangup was received on *fildes*.

[ETIME] A timeout occurred before an acknowledgement message was received at the stream head.

[EAGAIN] Temporarily unable to allocate storage to perform the linking operation.

[EINVAL] *arg* is an invalid multiplexor ID number.

[EINVAL] *fildes* is the file descriptor of a pipe.

An **I_PUNLINK ioctl** can also fail if it is waiting for the multiplexor to acknowledge the unlink request and an error (**M_ERROR**) message, or hangup (**M_HANGUP**) is received at the stream head for *fildes*. In addition, an error can be returned in an **M_IOCACK** or **M_IOCNAK** message. When these occur, the **P_UNLINK ioctl** fails with **errno** set to the value in the message.

I_PUSH

[EINVAL] An invalid module name was used.

[EFAULT] *arg* points outside the allocated address space.

[ENXIO] Error value returned by the module being pushed. The push has failed.

[ENXIO] A hangup was received on *fildes*.

I_RECVFD

[EAGAIN] The **O_NONBLOCK** option was set, and a message was not present on the stream head read queue.

[EFAULT] The *arg* parameter points outside the allocated address space.

[EBADMSG] The message present on the stream head read queue did not contain a passed file descriptor.

[EMFILE] Too many open files. No more file descriptors are permitted to be opened.

[ENXIO] A hangup was received on *fildes*.

I_SENDFD

[EAGAIN] The sending stream head could not allocate a message block for the file pointer.

[EAGAIN] The read queue of the receiving stream head was full and could not accept the message.

[EBADF] The *arg* parameter is not a valid open file descriptor.

[EINVAL] The *fildes* parameter does not refer to a stream.

[ENXIO] A hangup was received on *fildes*.

I_SETCLTIME

[EINVAL] *arg* has an illegal value.

I_SETSIG

[EINVAL] The user process is not registered to receive the **SIGPOLL** signal.

[EAGAIN] A data structure to store the signal request could not be allocated.

I_SRDOPT

[EINVAL] *arg* contains an illegal value.

I_STR

[EINVAL] The **ic_len** field is less than 0 (zero) bytes or larger than the maximum allowable size of the data part of a message (**ic_dp**).

[EINVAL] The **ic_timout** field is less than –1.

[EFAULT] *arg* points, or the buffer area specified by **ic_dp** or **ic_len** is, outside the allocated address space.

[ENOSR] Buffers could not be allocated for the **ioctl** request because of a lack of STREAMS memory resources.

[ENXIO] A hangup was received on the stream referred to by *fildes*.

[ETIME] The **ioctl** request timed out before an acknowledgement was received.

The **I_STR ioctl** can also fail if the stream head receives a message indicating an error (**M_ERROR**) or a hangup (**M_HANGUP**). In addition, an error can be returned in an **M_IOCACK** or **M_IOCNAK** message. In these cases, the **ioctl** fails with **errno** set to the error value in the message.

I_SWROPT

[EINVAL] The *arg* parameter is an illegal value.

I_UNLINK

[ENXIO] A hangup was received on *fildes*.

[ETIME] A timeout occurred before an acknowledgement message was received at the stream head.

[EINVAL] *arg* is an invalid multiplexor ID number, or *fildes* is already linked under a multiplexing driver.

An **I_UNLINK ioctl** can also fail if it is waiting for the multiplexor to acknowledge the unlink request and an error (**M_ERROR**) message, or hangup (**M_HANGUP**) is received at the stream head for *fildes*. In addition, an error can be returned in **M_IOCACK** or **M_IOCNAK** message. When this occurs, the **I_UNLINK ioctl** fails with **errno** set to the value in the message.

SEE ALSO

close(2), fcntl(2), getmsg(2), ioctl(2), open(2), poll(2), putmsg(2), read(2), write(2), signal(5).

NAME

strlog - STREAMS log driver

DESCRIPTION

The STREAMS log driver allows user-level processes and STREAMS drivers and modules to perform error logging and event tracing. These tasks are done via a user interface and a kernel interface. Further, the STREAMS log driver delivers error logging and event tracing messages to the Network Tracing and Logging Facility (NetTL) (see *nettl*(1M), *netfmt*(1M), and *nettlconf*(1M)).

The interface that this driver presents to user-level processes is a subset of the **ioctl()** system calls and STREAMS message formats. These processes can be error loggers, trace loggers, or other user processes, that generate error or event messages. The user interface collects log messages from the log driver, and also generates log messages from user processes.

The driver also accepts log messages from STREAMS drivers and modules in the kernel via its function call interface. The kernel interface enters requests or calls from STREAMS drivers and modules into log messages.

The log messages accepted by the log driver are also delivered to NetTL. NetTL can be used to control which types of messages to log, and to format and filter the logged messages.

Kernel Interface

STREAMS drivers and modules generate log messages by calls to the **strlog** function.

```
#include <sys/strlog.h>

int strlog (mid, sid, level, flags, fmt [, value ]...);
short mid;
short sid;
char level;
ushort flags;
char *fmt;
int value;
```

mid specifies the STREAMS module ID number for the driver or module submitting the log message.

sid specifies the sub-ID number of a minor device associated with the STREAMS module or driver identified by *mid*.

level specifies a level for screening lower-level event messages from a tracer.

flags contains several flags that can be set in various combinations. The flags are as follows:

- **SL_ERROR** The message is for the error logger.
- **SL_TRACE** The message is for the tracer.
- **SL_CONSOLE** The message will be printed to the console.
- **SL_FATAL** Provides a notification of a fatal error.
- **SL_NOTIFY** Makes a request to mail a copy of a message to the system administrator.

The following are additional flags. These flags are not used by **strerr** or **strace**. However, they are used to map STREAMS messages to NetTL messages as described below in *STREAMS-NetTL Link* section.

- **SL_WARN** The message is a warning.
- **SL_NOTE** The message is a note.

fmt is a **printf** style format string. This accepts the **%x**, **%l**, **%o**, **%u**, **%d**, **%c**, and **%s** conversion specifications.

values are numeric or character arguments for the format string. There is no maximum number of arguments that can be specified.

User Interface

User processes access the log driver with an **open()** call to **/dev/strlog**. Each open to the device will obtain a separate stream. After a process opens **/dev/strlog**, it indicates whether it is an error logger or trace logger. It does this by issuing an **I_STR ioctl()** system call with the appropriate value in the **ic_cmd** field of the **strioctl** structure, and the appropriate data and control information in a **trace_ids** structure:

```
struct trace_ids {
    short   ti_mid;
    short   ti_sid;
    char    ti_level;
    short   ti_flags;
};
```

The values for **ic_cmd** are:

I_ERRLOG Indicates an error logger. No **trace_ids** data is needed.

I_TRCLOG Indicates a trace logger. A data buffer consisting of an array of one or more **trace_ids** structures must be included.

If any of the fields of the **trace_ids** structure contain a value of -1, **/dev/strlog** will accept whatever value it receives in that field. Otherwise, **strlog** only accepts messages only if the values of *mid* and *sid* are the same as their counterparts in the **trace_ids** structure, and if the message's level is equal to or less than the **level** value in the **trace_ids** structure.

Once the logger process has sent the **I_STR ioctl()** call, the STREAMS log driver begins to send log messages matching the restrictions to the logger process. The logger process obtains the log messages via the **getmsg()** system call. The control part of the messages passed in this call includes a **log_ctl** structure:

```
struct log_ctl {
    short   mid;
    short   sid;
    char    level;
    short   flags;
    long    ltime;
    long    ttime;
    int     seq_no;
};
```

The **log_ctl** structure indicates the *mid*, *sid*, and *level* time in ticks since the boot time that the message was submitted, the corresponding time in seconds since January 1, 1970, and a sequence number. The time in seconds since January 1, 1970 is provided so that the date and time of the message can be easily computed. The time in ticks since boot time is provided so that the relative timing of log messages can be determined.

S

A user process, other than an error or trace logger, can send a log message to **strlog**. The driver will accept only the **flags** and **level** fields of the **log_ctl** structure in the control part of the message, and a properly formatted data part of the message. The data part of the message is properly formatted if it contains a null-terminated format string, followed by up to three arguments packed one word each after the end of the string.

A different series of sequence numbers is provided for error and trace logging streams. These sequence numbers are intended to help track the delivery of the messages. A gap in a sequence of numbers indicates that the logger process did not successfully deliver them. This can happen if the logger process stops sending messages for one reason or another (see *strace*(1M) and *strerr*(1M) command reference pages for more information). The data part of messages contains text of the format string (null terminated), followed by up to three arguments.

STREAMS-NetTL Link

Both STREAMS error logging and event tracing messages are mapped to NetTL logging messages, and are delivered to NetTL. NetTL classifies messages into four log classes: DISASTER, ERROR, WARNING, and INFORMATIVE. The NetTL log class is determined by the **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. This setting can be altered by the **nettl** command or the **nettlconf** command (see *nettl*(1M) and *nettlconf*(1M)).

The STREAMS subsystem ID used by NetTL is **STREAMS**.

The messages logged by NetTL facility can be formatted to a readable form by the **netfmt** command (see *netfmt*(1M)). The **netfmt** accepts a filter configuration file, which can be used to filter on STREAMS module ID and sub-ID. The filter configuration file syntax for STREAMS is the following:

STREAMS *module_id sub_id*

module_id and *sub_id* can be a decimal number or * as a wild card.

RETURN VALUE

Unless specified otherwise, upon successful completion, the **strlog ioctl()** commands return a value of 0 (zero). Otherwise, a value of -1 is returned.

ERRORS

If any of the following conditions occurs, **strlog** driver's **ioctl()** command sets **errno** to the corresponding value:

[ENXIO] The **I_TRCLOG ioctl()** call did not contain any **trace_ids** structures.

[ENXIO] The **I_STR ioctl()** call could not be recognized.

The driver does not return any errors for incorrectly formatted messages that user processes send.

EXAMPLES

The following examples illustrate some basic uses for the **strlog** interface.

This code example segment shows how a STREAMS module causes a message to be printed to the console:

```
strlog(TMUX,minor(mydev),0,SL_CONSOLE|SL_FATAL,
    "TMUX driver (minor:%d) suffers resource shortage.",
    minor(mydev));
```

This code example shows how a user process registers itself with the STREAMS log driver using the **ioctl()** command, **I_ERRLOG**.

```
struct strioctl iocerr:
int logfd;

if ((logfd = open("/dev/strlog", O_RDWR)) == -1) {
    printf("Cannot open /dev/strlog\n");
    exit(1);
}

iocerr.ic_cmd = I_ERRLOG;
iocerr.ic_timout = 0;
iocerr.ic_len = 0;
iocerr.ic_dp = NULL;
ioctl(logfd, I_STR, &iocerr);
```

This code example shows a user-level process sending a message to the **strlog** driver.

```
struct strbuf control, data;
struct log_ctl log;
char *warning = "Fatal error for user level process";
int logfd;

if ((logfd = open("/dev/strlog", O_RDWR)) == -1) {
        printf("Cannot open /dev/strlog\n");
        exit(1);
}

control.len = control.maxlen = sizeof(log);
```

S

```
control.buf = (char *)&lc;

data.len = data.maxlen = strlen(warning);
data.buf = warning;

lc.level = 2;
lc.flags = SL_FATAL|SL_CONSOLE;

putmsg(logfd, &control, &data, 0);
```

The following examples illustrate how to use the NetTL facility for the STREAMS. See *nettl*(1M), *netfmt*(1M), *nettlconf*(1M) for the general NetTL usage. The STREAMS subsystem ID used by NetTL is **STREAMS**.

The **netfmt** accepts a filter configuration file as a command argument. The following filter configuration file example is used to format the messages whose module ID is 1 and sub-ID is 100:

```
STREAMS    1    100
```

This filter configuration file example can be used to display all the messages whose module ID is 2 and all the messages whose sub-ID is 101:

```
STREAMS    2    *
STREAMS    *    101
```

FILES

/dev/strlog specifies the clone interface.

<sys/strlog.h> specifies the header file for streams logging.

<stropts.h> specifies the header file for STREAMS options and **ioctl()** commands.

SEE ALSO

netfmt(1M), nettl(1M), nettlconf(1M), strace(1M), strerr(1M), getmsg(2), ioctl(2), open(2), putmsg(2), write(2), clone(7), streamio(7).

S

NAME

sttyv6: stty - terminal interface for Version 6/PWB compatibility

DESCRIPTION

These routines attempt to map the UNIX Time-Sharing System, Sixth Edition (Version 6), and PWB **stty()** and **gtty()** calls into the current ioctls that perform the same functions. The mapping cannot be perfect. The way the features are translated is described below. The reader should be familiar with **termio** before studying this entry.

The following data structure is defined in the include file **<sgtty.h>**:

```
struct sgttyb {
     char  sg_ispeed;  /* input speed */
     char  sg_ospeed;  /* output speed */
     char  sg_erase;   /* erase character */
     char  sg_kill;    /* kill character */
     int  sg_flags;    /* mode flags */
}
```

The flags, as defined in **sgtty.h**, are:

HUPCL	01
XTABS	02
LCASE	04
ECHO	010
CRMOD	020
RAW	040
ODDP	0100
EVENP	0200
ANYP	0300
NLDELAY	001400
TBDELAY	002000
CRDELAY	030000
VTDELAY	040000
BSDELAY	0100000

CR0	0
CR1	010000
CR2	020000
CR3	030000
NL0	0
NL1	000400
NL2	001000
NL3	001400
TAB0	0
TAB1	002000
NOAL	004000
FF0	0
FF1	040000
BS0	0
BS1	0100000

When the **stty** command (*ioctl* **TIOCSETP**) is executed, the flags in the old **sgttyb** structure are mapped into their new equivalents in the **termio** structure. Then the **TCSETA** command is executed.

The following table shows the mapping between the old **sgttyb** flags and the current **termio** flags. Note that flags contained in the **termio** structure that are not mentioned below are cleared.

HUPCL (if set)	Sets the **termio** HUPCL flag.
HUPCL (if clear)	Clears the **termio** HUPCL flag.
XTABS (if set)	Sets the **termio** TAB3 flag.
XTABS (if clear)	Clears the **termio** TAB3 flag.
TBDELAY (if set)	Sets the **termio** TAB1 flag.

S

TBDELAY (if clear)	Clears the **termio** TAB1 flag.
LCASE (if set)	Sets the **termio** IUCLC, OLCUC, and XCASE flags.
LCASE (if clear)	Clears the **termio** IUCLC, OLCUC, and XCASE flags.
ECHO (if set)	Sets the **termio** ECHO flag.
ECHO (if clear)	Clears the **termio** ECHO flag.
NOAL (if set)	Sets the **termio** ECHOK flag.
NOAL (if clear)	Clears the **termio** ECHOK flag.
CRMOD (if set)	Sets the **termio** ICRNL and ONLCR flags; also, if CR1 is set, the **termio** CR1 flag is set, and if CR2 is set, the **termio** ONOCR and CR2 flags are set.
CRMOD (if clear)	sets the **termio** ONLRET flag; also, if NL1 is set, the **termio** CR1 flag is set, and if NL2 is set, the **termio** CR2 flag is set.
RAW (if set)	Sets the **termio** CS8 flag, and clears the **termio** ICRNL and IUCLC flags; also, default values of 6 characters and 0.1 seconds are assigned to MIN and TIME, respectively.
RAW (if clear)	Sets the **termio** BRKINT, IGNPAR, ISTRIP, IXON, IXANY, OPOST, CS7, PARENB, ICANON, and ISIG flags; also, the default values control-D and null are assigned to the control characters EOF and EOL, respectively.
ODDP (if set)	If EVENP is also set, clears the **termio** INPCK flag; otherwise, sets the **termio** PARODD flag.
VTDELAY (if set)	Sets the **termio** FFDLY flag.
VTDELAY (if clear)	Clears the **termio** FFDLY flag.
BSDELAY (if set)	Sets the **termio** BSDLY flag.
BSDELAY (if clear)	Clears the **termio** BSDLY flag.

In addition, the **termio** CREAD bit is set, and, if the baud rate is 110, the CSTOPB bit is set.

When using **TIOCSETP**, the *ispeed* entry in the **sgttyb** structure is mapped into the appropriate speed in the **termio** CBAUD field. The *erase* and *kill* **sgttyb** entries are mapped into the **termio** erase and kill characters.

When the **gtty** (*ioctl* **TIOCGETP**) command is executed, the **termio TCGETA** command is first executed. The resulting **termio** structure is then mapped into the **sgttyb** structure, which is then returned to the user.

The following table shows how the **termio** flags are mapped into the old **sgttyb** structure. Note that all flags contained in the **sgttyb** structure that are not mentioned below are cleared.

HUPCL (if set)	Sets the **sgttyb** HUPCL flag.
HUPCL (if clear)	Clears the **sgttyb** HUPCL flag.
ICANON (if set)	Sets the **sgttyb** RAW flag.
ICANON (if clear)	Clears the **sgttyb** RAW flag.
XCASE (if set)	Sets the **sgttyb** LCASE flag.
XCASE (if clear)	Clears the **sgttyb** LCASE flag.
ECHO (if set)	Sets the **sgttyb** ECHO flag.
ECHO (if clear)	Clears the **sgttyb** ECHO flag.
ECHOK (if set)	Sets the **sgttyb** NOAL flag.
ECHOK (if clear)	Clears the **sgttyb** NOAL flag.
PARODD (if set)	Sets the **sgttyb** ODDP flag.
PARODD (if clear)	Clears the **sgttyb** ODDP flag.
INPCK (if set)	Sets the **sgttyb** EVENP flag.

PARODD, INPCK (if both clear)
Sets the **sgttyb** ODDP and EVENP flags.

ONLCR (if set) Sets the **sgttyb** CRMOD flag; also, if CR1 is set, the **sgttyb** CR1 flag is set, and if CR2 is set, the **sgttyb** CR2 flag is set.

ONLCR (if clear) If CR1 is set, the **sgttyb** NL1 flag is set, and if CR2 is set, the **sgttyb** NL2 flag is set.

TAB3 (if set) Sets the **sgttyb** XTABS flag.

TAB3 (if clear) Clears the **sgttyb** XTABS flag.

TAB1 (if set) Sets the **sgttyb** TBDELAY flag.

TAB1 (if clear) Clears the **sgttyb** TBDELAY flag.

FFDLY (if set) Sets the **sgttyb** VTDELAY flag.

FFDLY (if clear) Clears the **sgttyb** VTDELAY flag.

BSDLY (if set) Sets the **sgttyb** BSDELAY flag.

BSDLY (if clear) Clears the **sgttyb** BSDELAY flag.

When using **TIOCGETP**, the **termio** CBAUD field is mapped into the *ispeed* and *ospeed* entries of the **sgttyb** structure. Also, the **termio** erase and kill characters are mapped into the *erase* and *kill* **sgttyb** entries.

Note that, since there is not a one-to-one mapping between the **sgttyb** and **termio** structures, unexpected results may occur when using the older **TIOCSETP** and **TIOCGETP** calls. Thus, the **TIOCSETP** and **TIOCGETP** calls should be replaced in all future code by the current equivalents, **TCSETA** and **TCGETA**, respectively.

WARNINGS

These facilities are included to aid in conversion of old programs, and should not be used in new code. Use the interface described in **termio**. Note that these conversions do *not* work for programs ported from UNIX Time-Sharing System, Seventh Edition (Version 7), because some V7 flags are defined differently.

SEE ALSO

stty(2), termio(7).

S

NAME

TCP - Internet Transmission Control Protocol

SYNOPSIS

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>

s = socket(AF_INET, SOCK_STREAM, 0);
s = socket(AF_INET6, SOCK_STREAM, 0);
```

DESCRIPTION

The TCP protocol provides reliable, flow-controlled, two-way transmission of data. It is a byte-stream protocol used to support the **SOCK_STREAM** socket type. TCP constructs virtual circuits between peer entities. A virtual circuit consists of remote Internet addresses, remote ports, local Internet addresses and local ports. IP uses the Internet addresses to direct messages between hosts, and the port numbers to identify a TCP entity at a particular host.

Sockets using TCP are either **active** or **passive**. **connect()** creates active sockets, which initiate connections to passive sockets (see *connect*(2)). To create a passive socket, use the **listen()** system call after binding the socket with the **bind()** system call (see *listen*(2) and *bind*(2)). Only passive sockets can use the **accept()** call to accept incoming connections (see *accept*(2)).

Passive sockets can *underspecify* their location to match incoming connection requests from multiple networks. This technique, called **wildcard addressing**, allows a single server to provide service to clients on multiple networks. To create a socket that listens on all networks, the Internet address **INADDR_ANY** must be bound for AF_INET family and **in6addr_any** for AF_INET6 family. The TCP port can still be specified even if wildcard addressing is being used. If the port is specified as zero, the system assigns a port.

Once **accept()** has a rendezvous with a connect request, a virtual circuit is established between peer entities. **bind()** supplies the local port and local Internet address and **accept()** gathers the remote port and remote Internet address from the peer requesting the connection.

Options

The system supports the following socket options: **TCP_MAXSEG**, **TCP_NODELAY**, **TCP_ABORT_THRESHOLD**, **TCP_CONN_ABORT_THRESHOLD**, **TCP_KEEPCNT**, **TCP_KEEPIDLE**, **TCP_KEEPINTVL**, **TCP_TSOPTENA**, and **TCP_SACKENA** (defined in the include file **<netinet/tcp.h>**). The **TCP_MAXSEG** option can only be used with **getsockopt()**, while **TCP_NODELAY**, **TCP_ABORT_THRESHOLD**, **TCP_CONN_ABORT_THRESHOLD** **TCP_KEEPCNT**, **TCP_KEEPIDLE**, **TCP_KEEPINTVL**, **TCP_TSOPTENA**, and **TCP_SACKENA** can be set with **setsockopt()** and tested with **getsockopt()** (see *getsockopt*(2)). These options require *level* to be set to **IPPROTO_TCP** in the **getsockopt/setsockopt** call.

TCP_MAXSEG (non-boolean option) lets an application to receive the current segment size of the TCP SOCK_STREAM socket. The current segment size will be returned in *optval*.

TCP_NODELAY
(boolean option) causes small amounts of output to be sent immediately.

TCP_ABORT_THRESHOLD
(non-boolean option) sets the second threshold timer for the connections that are in ESTABLISHED state. The option value is the threshold time in milliseconds.

When it must retransmit packets because a timer has expired, TCP first compares the total time it has waited against the two thresholds, as described in RFC 1122, 4.2.3.5. If it has waited longer than the second threshold (R2), TCP terminates the connection. The default value for this option is the current value of the ndd tunable parameter **tcp_ip_abort_interval**. Refer to *ndd*(1M) online help for details on the **tcp_ip_abort_interval** default value.

TCP_CONN_ABORT_THRESHOLD
(non-boolean option) sets the second threshold timer during connection establishment. The option value is the threshold time in milliseconds.

This option is the same as **TCP_ABORT_THRESHOLD**, except that this value is used during connection establishment. When it must retransmit the SYN packet because a timer has expired, TCP first compares the total time it has waited against the two thresholds. If it has waited longer than the second threshold, TCP terminates the connection. The default value for this option is the current value of the ndd tunable **tcp_ip_abort_cinterval**. See *ndd*(1M) online help for details on the **tcp_ip_abort_cinterval** default value.

TCP_KEEPCNT
(non-boolean option) When the **SO_KEEPALIVE** option is enabled, TCP probes a connection that has been idle for some amount of time. If the remote system does not respond to a keepalive probe, TCP retransmits the probe a certain number of times before a connection is considered to be broken. The **TCP_KEEPCNT** option can be used to affect this value for a given socket, and specifies the maximum number of keepalive probes to be sent. This option takes an **int** value, with a range of 1 to 32767.

TCP_KEEPIDLE
(non-boolean option) When the **SO_KEEPALIVE** option is enabled, TCP probes a connection that has been idle for some amount of time. The default value for this idle period is 2 hours. The **TCP_KEEPIDLE** option can be used to affect this value for a given socket, and specifies the number of seconds of idle time between keepalive probes. This option takes an **int** value, with a range of 1 to 32767.

TCP_KEEPINIT
(non-boolean option) If a TCP connection cannot be established within some amount of time, TCP will time out the connect attempt. The default value for this initial connection establishment timeout is 75 seconds. The **TCP_KEEPINIT** option can be used to affect this initial timeout period for a given socket, and specifies the number of seconds to wait before the connect attempt is timed out. For passive connections, the **TCP_KEEPINIT** option value is inherited from the listening socket. This option takes an **int** value, with a range of 1 to 32767.

TCP_KEEPINTVL
(non-boolean option) When the **SO_KEEPALIVE** option is enabled, TCP probes a connection that has been idle for some amount of time. If the remote system does not respond to a keepalive probe, TCP retransmits the probe after some amount of time. The default value for this retransmit interval is 75 seconds. The **TCP_KEEPINTVL** option can be used to affect this value for a given socket, and specifies the number of seconds to wait before retransmitting a keepalive probe. This option takes an **int** value, with a range of 1 to 32767.

TCP_TSOPTENA
(boolean option) When this option is enabled, the sender places a timestamp in each data segment. The receiver, if configured to accept them, sends these timestamps back in ACK segments. This provides the sender with a mechanism with which to measure round-trip time. TCP provides a Boolean option, **TCP_TSOPTENA** (from the **<netinet/tcp.h>** header file) to enable or disable this option. This option takes an **int** value. When this option is enabled, the **TCP_PAWS** option is also enabled.

TCP_PAWS
(boolean option) When the PAWS (Protect Against Wrapped Sequence numbers) option is enabled, the receiver rejects any old duplicate segments that are received. This option is used on synchronized TCP connections only. TCP provides a Boolean option, **TCP_PAWS** (from the **<netinet/tcp.h>** header file) to enable or disable this option. This option takes an **int** value. This option automatically turns the **TCP_TSOPTENA** option on.

TCP_SACKENA
(boolean option) When the Selective Acknowledgment (SACK) option is enabled, the data receiver can inform the sender about all segments that have arrived successfully. In this way, the sender need retransmit only those segments that have actually been lost. This option is useful in cases where multiple segments are dropped. TCP provides a Boolean option, **TCP_SACKENA** (from the **<netinet/tcp.h>** header file) to enable or disable this option. This option takes an **int** value.

If **TCP_NODELAY** is set, the system sends small amounts of output immediately rather than gathering them into a single packet after an acknowledgement is received. If **TCP_NODELAY** is not set, the system sends data when it is presented, if there is no outstanding unacknowledged data. If there is outstanding unacknowledged data, the system gathers small amounts of data to be sent in a single packet once an acknowledgement is received. For clients such as window managers that send a stream of mouse events which receive no replies, this packetization may cause significant delays. The **TCP_NODELAY** option can be used to avoid this situation. Note, however, that setting the **TCP_NODELAY** option may result in a large number of small packets being sent over the network.

By default, **TCP_NODELAY** is not set when a socket is created.

The option level to use for accessing the TCP option with the **setsockopt()** or **getsockopt()** calls is the protocol number for TCP which is available from **getprotobyname()** (see *getprotoent*(3N)).

If the **SO_KEEPALIVE** socket option is enabled on an established TCP connection and the connection has been idle for two hours, TCP sends a packet to the remote socket, expecting the remote TCP to acknowledge that the connection is still active. If the remote TCP does not respond in a timely manner, TCP continues to send keepalive packets according to its normal retransmission algorithm. If the remote TCP does not respond within a particular time limit, TCP drops the connection. The next socket system call (for example, **recv()**) returns an error, and **errno** is set to [ETIMEDOUT]. See *getsockopt*(2) for details on enabling **SO_KEEPALIVE**.

The default send and receives buffer size is 32768 bytes (see *WARNINGS* below). The send and receive buffer sizes for TCP stream sockets can be altered by using the **SO_SNDBUF** and **SO_RCVBUF** options of the **setsockopt()** system call or the **XTI_SNDBUF** and **XTI_RCVBUF** options of the **t_optmgmt()** system call. Refer to *getsockopt*(2) or *t_optmgmt*(3) for details.

The maximum transmit buffer size for a TCP stream socket is 2147483647 bytes. The maximum receive buffer size for a TCP stream socket is 1073725440 bytes. These maximum values can be lowered using the ndd variables **tcp_xmit_hiwater_max** and **tcp_recv_hiwater_max**.

ERRORS

One of the following errors may be returned in **errno** if a socket operation fails. For a more detailed list of errors, see the man pages for specific system calls.

[EISCONN] The socket is already connected.

[ENOBUFS] No buffer space is available for an internal data structure.

[ETIMEDOUT]
Connection dropped due to excessive retransmissions.

[ECONNRESET]
The connection was forcibly closed by the peer socket.

[ECONNREFUSED]
Remote peer actively refuses connection establishment (usually because no process is listening to the port).

[EADDRINUSE]
The specified address is already in use.

[EADDRNOTAVAIL]
The specified address is not available on this machine.

WARNINGS

The default socket buffer size might increase without notice in a future release or patch. Therefore, if an application calls **setsockopt()** with **SO_RCVBUF**, it should do so before calling **listen()**, or it should first call **getsockopt()** with **SO_RCVBUF** and ensure that the intended new receive buffer size is not less than the current buffer size. These programming conventions are consistent with TCP protocol restrictions against reducing the TCP receive window after a connection has been established.

AUTHOR

The socket interfaces to TCP were developed by the University of California, Berkeley.

SEE ALSO

ndd(1M), getsockopt(2), recv(2), send(2), socket(2), t_open(3), t_optmgmt(3), socket(7), inet(7F).

RFC 793 Transmission Control Protocol
RFC 1122 Requirements for Internet hosts
RFC 1323 TCP Extensions for High Performance

RFC 1878	Variable Length Subnet Table for IPv4
RFC 2018	TCP Selective Acknowledgement Options
RFC 2414	Increasing TCP's Initial Window
RFC 2582	NewReno Modifications to TCP's Fast Recovery Algorithm

NAME

tels, telm - STREAMS Telnet slave (pseudo-terminal) driver, STREAMS Telnet master driver (used by telnetd only), respectively

SYNOPSIS

```
#include <sys/termios.h>
#include <sys/strtio.h>

int open("/dev/pts/tN", O_RDWR);
```

DESCRIPTION

A Telnet pseudo-terminal consists of a tightly-coupled pair of character devices, called the master device and slave device. The master and slave device drivers work together to provide a Telnet connection on the server side where the master provides a connection to **telnetd** and the slave provides a terminal device special file access for the Telnet application processes, as depicted below:

```
                          --------------------------
                         | Pseudo terminal functions|
    Application <-->     |--------------------------| <--> telnetd
     Processes           | Slave        |   Master   |
                         | (tels)       |   (telm)   |
                          --------------------------
```

The slave driver, **tels** with **ptem** (STREAMS pty emulation module) and **ldterm** (STREAMS line discipline module) pushed on top (not shown for simplicity), provides a terminal interface as described in *termio*(7). Whereas devices that provide the terminal interface described in *termio*(7) have a hardware device behind them; in contrast, the slave device has **telnetd** manipulating it through the master side of the Telnet pseudo terminal.

There are no nodes in the file system for each individual master device. Rather, the master driver is set up as a STREAMS *clone*(7) driver with its major device number set to the major for the clone driver and its minor device number set to the major for the **telm** driver. The master driver is opened by telnetd using the *open*(2) system call with **/dev/telnetm** as the device file parameter. The clone open finds the next available minor number for the master device. The master device is available only if it and its corresponding slave device are not already opened.

In order to use the STREAMS Telnet subsystem, a node for the master driver **/dev/telnetm** and *N* number of Telnet slave devices must be installed.

The number of slave devices is set by a kernel tunable parameter called **nstrtel**. This can be modified using SAM; its default and minimum value is 60. The value of **nstrtel** is the upper limit of the number of telnet sessions that can be opened.

Multiple opens are allowed on the Telnet slave device.

The master and slave drivers pass all STREAMS messages to their adjacent drivers. When the connection is closed from the Telnet client side, an **M_HANGUP** message is sent to the corresponding slave device which will render that slave device unusable. The process on the slave side gets the errno **ENXIO** when attempting a *write*(2) system call to the slave device file but it will be able to read any data remaining in the slave stream. Finally, when all the data has been read, the *read*(2) system call will return 0, indicating that the slave can no longer be used.

t

AUTHOR

tels() and **telm()** were developed by HP.

FILES

/dev/telnetm Streams Telnet master clone device

/dev/pts/tN Streams slave devices where *N* is the minor number of the slave device and 0 < *N* < **nstrtel**.

SEE ALSO

insf(1M), open(2), ioctl(2), streamio(7), ldterm(7), telnetd(1M), ptem(7).

NAME

termio, termios - general terminal interface

DESCRIPTION

All HP-UX asynchronous communications ports use the same general interface, regardless of what hardware is involved. Network connections such as **rlogin** (see *rlogin*(1) use the pseudo-terminal interface (see *pty*(7).

This discussion centers around the common features of this interface.

Opening a Terminal File

When a terminal file is opened, it normally causes the process to wait until a connection is established. In practice, users' programs seldom open these files; they are opened by special programs such as **getty** (see *getty*(1M)) and become a user's standard input, standard output, and standard error files.

If both the **O_NDELAY** and **O_NONBLOCK** flags (see *open*(2)) are clear, an *open* blocks until the type of modem connection requested (see *modem*(7)) is completed. If either the **O_NDELAY** or **O_NONBLOCK** flag is set, an *open* succeeds and return immediately without waiting for the requested modem connection to complete. The **CLOCAL** flag (see *Control Modes*) can also affect *open*(2).

Process Groups

A terminal can have a foreground process group associated with it. This foreground process group plays a special role in handling signal-generating input characters.

Command interpreter processes can allocate the terminal to different *jobs* (process groups) by placing related processes in a single process group and associating this process group with the terminal. A terminal's foreground process group can be set or examined by a process, assuming that the permission requirements are met (see *tcsetpgrp*(3C) or *tcgetpgrp*(3C)). The terminal interface aids in this allocation by restricting access to the terminal by processes that are not in the foreground process group.

A process group is considered orphaned when the parent of every member of the process group is either itself a member of the process group or is not a member of the group's session (see *Sessions*).

Sessions

A process that creates a session (see *setsid*(2) or *setpgrp*(2)) becomes a session leader. Every process group belongs to exactly one session. A process is considered to be a member of the session of which its process group is a member. A newly created process joins the session of its parent. A process can change its session membership (see *setpgid*(2) or *setpgrp*(2)). Usually a session comprises all the processes (including children) created as a result of a single login.

The Controlling Terminal

A terminal can belong to a process as its controlling terminal. Each process of a session that has a controlling terminal has the same controlling terminal. A terminal can be the controlling terminal for at most one session. The controlling terminal for a session is allocated by the session leader. If a session leader has no controlling terminal and opens a terminal device file that is not already associated with a session without using the **O_NOCTTY** option (see *open*(2), the terminal becomes the controlling terminal of the session and the controlling terminal's foreground process group is set to the process group of the session leader. While a controlling terminal is associated with a session, the session leader is said to be the controlling process of the controlling terminal.

The controlling terminal is inherited by a child process during a **fork()** (see *fork*(2)). A process relinquishes its controlling terminal if it creates a new session with **setsid()** or **setpgrp()** (see *setsid*(2) and *setpgrp*(2)), or when all file descriptors associated with the controlling terminal have been closed.

When the controlling process terminates, the controlling terminal is disassociated from the current session, allowing it to be acquired by a new session leader. A **SIGHUP** signal is sent to all processes in the foreground process group of the controlling terminal. Subsequent access to the terminal by other processes in the earlier session can be denied (see *Terminal Access Control*) with attempts to access the terminal treated as if a modem disconnect had been sensed.

Terminal Access Control

Read operations are allowed (see *Input Processing and Reading Data*) from processes in the foreground process group of their controlling terminal. If a process is not in the foreground process group of its controlling terminal, the process and all member's of its process group are considered to be in a background process group of this controlling terminal. All attempts by a process in a background process group to read from its controlling terminal will be denied. If denied and the reading process is ignoring or blocking the **SIGTTIN** signal, or the process (on systems that implement *vfork* separately from *fork*) has made a call to *vfork*(2) but has not yet made a call to *exec*(2), or the process group of the reading process is orphaned, **read()** returns −1 with **errno** set to **EIO** and no signal is sent. In all other cases where the read is denied, the process group of the reading process will be sent a **SIGTTIN** signal. The default action of the **SIGTTIN** signal is to stop the process to which it is sent.

If the process is in the foreground process group of its controlling terminal, write operations are allowed (see *Writing Data and Output Processing*). Attempts by a process in a background process group to write to its controlling terminal are denied if **TOSTOP** (see *Local Modes*) is set, the process is not ignoring and not blocking the **SIGTTOU** signal, and the process (on systems that implement *vfork* separately from *fork*) has not made a call to *vfork*(2) without making a subsequent call to *exec*(2). If the write is denied and the background process group is orphaned, the **write()** returns −1 with **errno** set to **EIO**. If the write is denied and the background process group is not orphaned, the **SIGTTOU** signal is sent to the process group of the writing process. The default action of the **SIGTTOU** signal is to stop the process to which it is sent.

Certain calls that set terminal parameters are treated in the same fashion as write, except that **TOSTOP** is ignored; that is, the effect is identical to that of terminal writes when **TOSTOP** is set.

Input Processing and Reading Data

A terminal device associated with a terminal device file can operate in full-duplex mode, so that data can arrive, even while data output is occurring. Each terminal device file has an *input queue* associated with it into which incoming data is stored by the system before being read by a process. The system imposes a limit, **MAX_INPUT**, on the number of characters that can be stored in the input queue. This limit is dependent on the particular implementation, but is at least 256. When the input limit is reached, all saved characters are discarded without notice.

All input is processed either in canonical mode or non-canonical mode (see *Canonical Mode Input Processing* and *Non-Canonical Mode Input Processing*). Additionally, input characters are processed according to the **c_iflag** (see *Input Modes*) and **c_lflag** (see *Local Modes*) fields. For example, such processing can include *echoing*, which in general means transmitting input characters immediately back to the terminal when they are received from the terminal. This is useful for terminals that operate in full-duplex mode.

The manner in which data is provided to a process reading from a terminal device file depends on whether the terminal device file is in canonical or non-canonical mode.

Another dependency is whether the **O_NONBLOCK** or **O_NDELAY** flag is set by either *open*(2) or *fcntl*(2). If the **O_NONBLOCK** and **O_NDELAY** flags are both clear, the read request is blocked until data is available or a signal is received. If either the **O_NONBLOCK** or **O_NDELAY** flag is set, the read request completes without blocking in one of three ways:

- If there is enough data available to satisfy the entire request, **read()** completes successfully, having read all of the data requested, and returns the number of characters read.
- If there is not enough data available to satisfy the entire request, **read()** completes successfully, having read as much data as possible, and returns the number of characters read.
- If there is no data available, **read()** returns −1, with **errno** set to **EAGAIN** when the **O_NONBLOCK** flag is set. Otherwise, (flag **O_NONBLOCK** is clear and **O_NDELAY** is set) **read()** completes successfully, having read no data, and returns a count of 0.

The availability of data depends upon whether the input processing mode is canonical or non-canonical. The following sections, *Canonical Mode Input Processing* and *Non-Canonical Mode Input Processing*, describe each of these input processing modes.

Canonical Mode Input Processing (Erase and Kill Processing)

In canonical mode input processing, terminal input is processed in units of lines, where a line is delimited by a new-line (NL) character, an end-of-file (EOF) character, or an end-of-line character (EOL) or (EOL2). See *Special Characters* for more information on **NL**, **EOF**, **EOL**, and **EOL2**. This means that a read request does not return until an entire line has been typed or a signal has been received. Also, no matter how many characters are requested in the read call, at most one line will be returned. It is not, however, necessary to read a whole line at once; any number of characters can be requested in a read, even one, without losing information.

MAX_CANON is the limit on the number of characters in a line. This limit varies with each particular implementation, but is at least 256.

When the **MAX_CANON** limit is reached, all characters in the current undelimited line are discarded without notice.

Erase and kill processing occur when any of three special characters, the ERASE, WERASE, or KILL characters (see *Special Characters*), is received. This processing affects data in the input queue that has not yet been delimited by a NL, EOF, EOL, or EOL2 character. This undelimited data makes up the current line. The ERASE character deletes the last character in the current line, if one exists. The WERASE character deletes the last word in the current line, if one exists. A word is defined as a series of non-blank characters (tabs are equivalent to blanks). The KILL character deletes all data in the current line, if any, and optionally outputs a new-line (NL) character. These characters operate on a key-stroke basis, independent of any backspacing or tabbing that may have preceded them. ERASE, WERASE, and KILL characters have no effect if the current line is empty. ERASE, WERASE, and KILL characters are not placed in the input queue.

Non-Canonical Mode Input Processing (MIN/TIME Interaction)

In non-canonical mode input processing, input characters are not assembled into lines, and erase and kill processing does not occur. The values of the **MIN** and **TIME** members of the **c_cc** array (see *termios Structure*) are used to determine how to process the characters received. **MIN** represents the minimum number of characters that should be received before **read()** successfully returns. TIME is a timer of 0.10 second granularity that is used to timeout bursty and short term data transmissions. The four possible cases for MIN and TIME and their interactions are described below.

Case A: MIN > 0, TIME > 0

In this case, TIME serves as an inter-character timer and is activated after the first character is received. Since it is an inter-character timer, it is reset after each character is received. The interaction between MIN and TIME is as follows:

- As soon as one character is received, the inter-character timer is started.
- If MIN characters are received before the inter-character timer expires (remember that the timer is reset upon receipt of each character), the read is satisfied. If the timer expires before MIN characters are received, the characters received to that point are returned to the user.
- Note that if TIME expires, at least one character will be returned because the timer would not have been enabled unless a character was received. In this case (MIN > 0, TIME > 0) the read blocks until the MIN and TIME mechanisms are activated by the receipt of the first character, or a signal is received.

Case B: MIN > 0, TIME = 0

In this case, since the value of TIME is zero, the timer plays no role and only MIN is significant. A pending read is not satisfied until MIN characters are received after any previous read completes (that is, the pending read blocks until MIN characters are received), or a signal is received. A program that uses this case to handle record-based terminal I/O can block indefinitely in the read operation.

Case C: MIN = 0, TIME > 0

In this case, since the value of MIN is zero, TIME no longer represents an inter-character timer. It now serves as a read timer that is activated as soon as the **read()** function is processed. A read is satisfied as soon as a single character is received *or* the read timer expires. If the timer expires, no character is returned. If the timer does not expire, the only way the read can be satisfied is by a character being received. A read cannot block indefinitely waiting for a character because if no character is received within TIME × 0.10 seconds after the read is initiated, **read()** returns a value of zero, having read no data.

Case D: MIN = 0, TIME = 0

The number of characters requested or the number of characters currently available, whichever is less, is returned without waiting for more characters to be input. If no characters are available, **read()** returns a value of zero, having read no data.

Some points to note about MIN and TIME:

1. In the above explanations, the interactions of MIN and TIME are not symmetric. For example, when MIN > 0 and TIME = 0, TIME has no effect. However, in the opposite case where MIN = 0 and TIME > 0, both MIN and TIME play a role in that MIN is satisfied with the receipt of a single character.
2. Also note that in case A (MIN > 0, TIME > 0), TIME represents an inter-character timer while in case C (MIN = 0, TIME > 0), TIME represents a read timer.

t

These two points highlight the dual purpose of the MIN/TIME feature. Cases A and B (where MIN > 0) exist to handle burst mode activity (such as file transfer programs) where a program would like to process at least MIN characters at a time. In case A, the inter-character timer is activated by a user as a safety measure while in case B it is turned off.

Cases C and D exist to handle single character timed transfers. These cases are readily adaptable to screen-based applications that need to know if a character is present in the input queue before refreshing the screen. In case C the read is timed, while in case D it is not.

Another important note is that MIN is always just a minimum. It does not denote a record length. For example, if a program initiates a read of 20 characters when MIN is 10 and 25 characters are present, 20 characters will be returned to the user. Had the program requested all characters, all 25 characters would be returned to the user.

Furthermore, if TIME is greater than zero and MIN is greater than **MAX_INPUT**, the read will never terminate as a result of MIN characters being received because all the saved characters are discarded without notice when **MAX_INPUT** is exceeded. If TIME is zero and MIN is greater than **MAX_INPUT**, the read will never terminate unless a signal is received.

Special Characters

Certain characters have special functions on input, output, or both. Unless specifically denied, each special character can be changed or disabled. To disable a character, set its value to **_POSIX_VDISABLE** (see *unistd*(5)). These special functions and their default character values are:

INTR — (Rubout or ASCII DEL) special character on input and is recognized if **ISIG** (see *Local Modes*) is enabled. Generates a **SIGINT** signal which is sent to all processes in the foreground process group for which the terminal is the controlling terminal. Normally, each such process is forced to terminate, but arrangements can be made to either ignore or hold the signal, or to receive a trap to an agreed-upon location; see *signal*(2) and *signal*(5). If **ISIG** is set, the INTR character is discarded when processed. If **ISIG** is clear, the INTR character is processed as a normal data character, and no signal is sent.

QUIT — (Control-\ or ASCII FS) special character on input. Recognized if **ISIG** (see *Local Modes*) is set. The treatment of this character is identical to that of the INTR character except that a **SIGQUIT** signal is generated and the processes that receive this signal are not only terminated, but a core image file (called **core**) is created in the current working directory if the implementation supports core files.

SWTCH — (ASCII NUL) special character on input and is only used by the shell layers facility *shl*(1). The shell layers facility is not part of the general terminal interface. No special functions are performed by the general terminal interface when SWTCH characters are encountered.

ERASE — (**#**) special character on input and is recognized if **ICANON** (see *Local Modes*) is enabled. Erases the preceding character. Does not erase beyond the start of a line, as delimited by a NL, EOF, EOL, or EOL2 character. If **ICANON** is enabled, the ERASE character is discarded when processed. If **ICANON** is not enabled, the ERASE character is treated as a normal data character.

WERASE — (disabled) special character on input and is recognized if **ICANON** (see *Local Modes*) is enabled. Erases the preceding word. Does not erase beyond the start of a line, as delimited by a NL, EOF, EOL, or EOL2 character. If **ICANON** is enabled, the WERASE character is discarded when processed. If **ICANON** is not enabled, the WERASE character is treated as a normal data character.

KILL — (**@**) special character on input and is recognized if **ICANON** is enabled. KILL deletes the entire line, as delimited by a NL, EOF, EOL, or EOL2 character. If **ICANON** is enabled, the KILL character is discarded when processed. If **ICANON** is not enabled, the KILL character is treated as a normal data character.

EOF — (Control-D or ASCII EOT) special character on input and is recognized if **ICANON** is enabled. EOF can be used to generate an end-of-file from a terminal. When received, all the characters waiting to be read are immediately passed to the program without waiting for a new-line, and the EOF is discarded. Thus, if there are no characters waiting, (that is, the EOF occurred at the beginning of a line) a character count of zero is returned from **read()**, representing an end-of-file indication. If **ICANON** is enabled, the EOF character is discarded when processed. If **ICANON** is not enabled, the EOF character is treated as a normal data character.

NL (ASCII LF) special character on input and is recognized if **ICANON** flag is enabled. It is the line delimiter (**\n**). If **ICANON** is not enabled, the NL character is treated as a normal data character.

EOL (ASCII NUL) special character on input and is recognized if **ICANON** is enabled. EOL is an additional line delimiter similar to NL. It is not normally used. If **ICANON** is not enabled, the EOL character is treated as a normal data character.

EOL2 (disabled) special character on input and is recognized if **ICANON** is enabled. EOL2 is an additional line delimiter similar to EOL. It is not normally used. If **ICANON** is not enabled, the EOL2 character is treated as a normal data character.

SUSP (disabled) special character recognized on input. If **ISIG** is enabled, receipt of the SUSP character causes a **SIGTSTP** signal to be sent to all processes in the foreground process group for which the terminal is the controlling terminal, and the SUSP character is discarded when processed. If **ISIG** is not enabled, the SUSP character is treated as a normal data character. Command interpreter processes typically set SUSP to Control-Z.

DSUSP (disabled) special character recognized on input. If **ISIG** is enabled, and a process in the foreground process group attempts to read the DSUSP character, a **SIGTSTP** signal is sent to all processes in the foreground process group for which the terminal is the controlling terminal, and the DSUSP character is then discarded. If **ISIG** is not enabled, the DSUSP character is treated as a normal data character. Note that DSUSP is similar to SUSP except that the signal is sent when a process in the foreground process group attempts to read the DSUSP character, rather than when it is typed.

STOP (Control-S or ASCII DC3) special character on both input and output. If **IXON** (output control) is enabled, processing of the STOP character temporarily suspends output to the terminal device. This is useful with CRT terminals to prevent output from disappearing before it can be read. While output is suspended and **IXON** is enabled, STOP characters are ignored and not read. If **IXON** is enabled, the STOP character is discarded when processed. If **IXON** is not enabled, the STOP character is treated as a normal data character. If **IXOFF** (input control) is enabled, the system sends a STOP character to the terminal device when the number of unread characters in the input queue is approaching a system specified limit. This is an attempt to prevent this buffer from overflowing by telling the terminal device to stop sending data.

START (Control-Q or ASCII DC1) special character on both input and output. If **IXON** (output control) is enabled, processing of the START character resumes output that has been suspended. While output is not suspended and **IXON** is enabled, START characters are ignored and not read. If **IXON** is enabled, the START character is discarded when processed. If **IXON** is not enabled, the START character is treated as a normal data character. If IXOFF (input control) is enabled, the system sends a START character to the terminal device when the input queue has drained to a certain system-defined level. This occurs when the input queue is no longer in danger of possibly overflowing.

t

CR (ASCII CR) special character on input is recognized if **ICANON** is enabled. When **ICANON** and **ICRNL** are enabled and **IGNCR** is not enabled, this character is translated into a NL, and has the same affect as the NL character. If **ICANON** and **IGNCR** are enabled, the CR character is ignored. If **ICANON** is enabled and both **ICRNL** and **IGNCR** are not enabled, the CR character is treated as a normal data character.

LNEXT (disabled) special character recognized on input. Causes the special meaning of the next character to be ignored. This works for all special characters specified above. It allows characters to be input that would otherwise be interpreted by the system for a special function.

The special characters are assigned their default character values when the terminal port is opened. The default values used are those specified by the System V Interface Definition, Third Edition (SVID3), except for the WERASE (Control-W) and LNEXT (Control-V) characters which are set to **_POSIX_VDISABLE** to maintain binary compatibility with previous releases of HP-UX. The default character values assigned when the port is opened can be changed for all ports on a system wide basis through the use of the **stty** command (see *stty*(1)). The character values may also be changed for a specific port after it is opened using the **stty** command. The NL and CR characters cannot be changed or disabled. The character values for the remaining special characters can be changed or disabled to suit individual tastes.

If **ICANON** is set (see *Local Modes*), the ERASE, KILL, and EOF characters can be escaped by a preceding **** character, in which case no special function is performed. These characters, and the remaining special characters, may also be escaped by preceding them with the LNEXT character (see LNEXT above).

If two or more special characters have the same value, the function performed when the character is processed is undefined.

Modem Disconnect

If a modem disconnect is detected by the terminal interface for a controlling terminal, and if **CLOCAL** is clear in the **c_cflag** field for the terminal (see *Control Modes*), the **SIGHUP** signal is sent to the controlling process of the controlling terminal. Unless other arrangements have been made, this causes the controlling process to terminate. Any subsequent read from the terminal device returns with an end-of-file indication until the device is closed. Thus, processes that read a terminal file and test for end-of-file can terminate appropriately after a disconnect. Any subsequent **write()** to the terminal device returns –1, with **errno** set to **EIO**, until the device is closed.

Closing a Terminal Device File

The last process to close a terminal device file causes any output not already sent to the device to be sent to the device even if output was suspended. This last close always blocks (even if non-blocking I/O has been specified) until all output has been sent to the terminal device. Any input that has been received but not read is discarded.

Writing Data and Output Processing

When characters are written, they are placed on the output queue. Characters on the output queue are transmitted to the terminal as soon as previously-written characters are sent. These characters are processed according to the **c_oflag** field (see *Output Modes*). Input characters are echoed by putting them in the output queue as they arrive. If a process produces characters for output more rapidly than they can be sent, the process is suspended when its output queue exceeds some limit. When the queue has drained down to some threshold, the process is resumed.

termios Structure

Routines that need to control certain terminal I/O characteristics can do so by using the **termios** structure as defined in the header file **<termios.h>**. The structure is defined as follows:

```
#define NCCS      16
struct  termios   {
        tcflag_t  c_iflag;     /* input modes */
        tcflag_t  c_oflag;     /* output modes */
        tcflag_t  c_cflag;     /* control modes */
        tcflag_t  c_lflag;     /* local modes */
        tcflag_t  c_reserved;  /* reserved for future use */
        cc_t      c_cc[NCCS];  /* control chars */
};
```

t

The special characters are defined by the array **c_cc**. The relative positions and default values for each special character function are as follows:

INTR	VINTR	DEL
QUIT	VQUIT	Control-|
ERASE	VERASE	#
KILL	VKILL	@
EOF	VEOF	Control-D
EOL	VEOL	NUL
EOL2	VEOL2	disabled
MIN	VMIN	NUL
TIME	VTIME	Control-D
SUSP	VSUSP	disabled

START	VSTART	Control-Q
STOP	VSTOP	Control-S
WERASE	VWERASE	disabled
LNEXT	VLNEXT	disabled
DSUSP	VDSUSP	disabled

termio Structure

The **termio** structure has been superseded by the **termios** structure and is provided for backward compatibility with prior applications (see *termio Caveats*). The structure is defined in the header file **<termio.h>** and is defined as follows:

```
#define NCC     8
struct  termio  {
        unsigned short  c_iflag;    /* input modes */
        unsigned short  c_oflag;    /* output modes */
        unsigned short  c_cflag;    /* control modes */
        unsigned short  c_lflag;    /* local modes */
        char            c_line;     /* line discipline */
        unsigned char   c_cc[NCC];  /* control chars */
};
```

Modes

The next four sections describe the specific terminal characteristics that can be set using the **termios** and **termio** structures (see *termio Caveats*). Any bits in the modes fields that are not explicitly defined below are ignored. However, they should always be clear to prevent future compatibility problems.

Input Modes

The **c_iflag** field describes the basic terminal input control:

IGNBRK	Ignore break condition.
BRKINT	Signal interrupt on break.
IGNPAR	Ignore characters with parity errors.
PARMRK	Mark parity errors.
INPCK	Enable input parity check.
ISTRIP	Strip character.
INLCR	Map NL to CR on input.
IGNCR	Ignore CR.
ICRNL	Map CR to NL on input.
IUCLC	Map uppercase to lowercase on input.
IXON	Enable start/stop output control.
IXANY	Enable any character to restart output.
IXOFF	Enable start/stop input control.
IMAXBEL	Enable BEL on input line too long.

A break condition is defined as a sequence of zero-value bits that continues for more than the time to send one character. For example, a character framing or parity error with data all zeros is interpreted as a single break condition.

If **IGNBRK** is set, the break condition is ignored. Therefore the break condition cannot be read by any process. If **IGNBRK** is clear and **BRKINT** is set, the break condition flushes both the input and output queues and, if the terminal is the controlling terminal of a foreground process group, the break condition generates a single **SIGINT** signal to that foreground process group. If neither **IGNBRK** nor **BRKINT** is set, a break condition is read as a single **\0** character, or if **PARMRK** is set, as the three-character sequence **\377**, **\0**, **\0**.

If **IGNPAR** is set, characters with other framing and parity errors (other than break) are ignored.

If **PARMRK** is set, and **IGNPAR** is clear, a character with a framing or parity error (other than break) is read as the three-character sequence: **\377**, **\0**, *X*, where *X* is the data of the character received in error. To avoid ambiguity in this case, if **ISTRIP** is clear, a valid character of **\377** is read as **\377**, **\377**. If both **PARMRK** and **IGNPAR** are clear, a framing or parity error (other than break) is read as the character **\0**.

If **INPCK** is set, input parity checking is enabled. If **INPCK** is clear, input parity checking is disabled. Whether input parity checking is enabled or disabled is independent of whether parity detection is enabled or disabled (see *Control Modes*). If **PARENB** is set (see *Control Modes*) and **INPCK** is clear, parity

t

generation is enabled but input parity checking is disabled; the hardware to which the terminal is connected will recognize the parity bit, but the terminal special file will not check whether this bit is set correctly or not.

The following table shows the interrelationship between the flags **IGNBRK**, **BRKINT**, **IGNPAR**, and **PARMRK**. The column marked **Input** gives various types of input characters received, indicated as follows:

- **0** NUL character (**\0**)
- **C** Character other than NUL
- **P** Parity error detected
- **F** Framing error detected

Items enclosed in brackets indicate one or more of the conditions are true.

If the **INPCK** flag is clear, characters received with parity errors are not processed according to this table, but instead, as if no parity error had occurred. Under the flag columns, **Set** indicates the flag is set, **Clear** indicates the flag is not set, and **X** indicates the flag may be set or clear. The column labeled **Read** shows the results that will be passed to the application code. A — indicates that no character or condition is passed to the application code. The value **SIGINT** indicates that no character is returned, but that the **SIGINT** signal is sent to the foreground process group of the controlling terminal.

Input	IGNBRK	BRKINT	IGNPAR	PARMRK	Read
0[PF]	Set	X	X	X	—
0[PF]	Clear	Set	X	X	**SIGINT**
0[PF]	Clear	Clear	X	Set	'\377','\0','\0'
0[PF]	Clear	Clear	X	Clear	'\0'
C[PF]	X	X	Set	X	—
C[PF]	X	X	Clear	Set	'\377','\0',C
C[PF]	X	X	Clear	Clear	'\0'
'\377'	X	X	X	Set	'\377','\377'

If **ISTRIP** is set, valid input characters are first stripped to 7-bits, otherwise all 8-bits are processed.

If **INLCR** is set, a received NL character is translated into a CR character. If **IGNCR** is set, a received CR character is ignored (not read). If **IGNCR** is clear and **ICRNL** is set, a received CR character is translated into a NL character.

If **IUCLC** is set, a received uppercase alphabetic character is translated into the corresponding lowercase character.

If **IXON** is set, start/stop output control is enabled. A received STOP character suspends output and a received START character restarts output. If **IXANY** and **IXON** are set, any input character without a framing or parity error restarts output that has been suspended. When these three flags are set, output suspended, and an input character received with a framing or parity error, output resumes if processing it results in data being read. When **IXON** is set, START and STOP characters are not read, but merely perform flow control functions. When **IXON** is clear, the START and STOP characters are read.

If **IXOFF** is set, start/stop input control is enabled. The system transmits a STOP character when the number of characters in the input queue exceeds a system defined value (high water mark). This is intended to cause the terminal device to stop transmitting data in order to prevent the number of characters in the input queue from exceeding **MAX_INPUT**. When enough characters have been read from the input queue that the number of characters remaining is less than another system defined value (low water mark), the system transmits a START character which is intended to cause the terminal device to resume transmitting data (without risk of overflowing the input queue). In order to avoid potential deadlock, **IXOFF** is ignored in canonical mode whenever there is no line delimiter in the input buffer. In this case, the STOP character is not sent at the high water mark, but will be transmitted later if a delimiter is received. If all complete lines are read from the input queue leaving only a partial line with no line delimiter, the START character is sent, even if the number of characters is still greater than the low water mark. When **ICANON** is set and the input stream contains more characters between line delimiters than the high water mark allows, there is no guarantee that **IXOFF** can prevent buffer overflow and data loss, because the STOP character may not be sent in time, if at all.

If **IMAXBEL** is set, the ASCII BEL character is echoed if the input queue overflows. Further input is not stored, but any input present in the input queue is not discarded. If **IMAXBEL** is clear, no ASCII BEL character is echoed, and the input already present in the input queue is discarded when the input queue overflows.

The initial input control value is all bits clear.

Output Modes

The **c_oflag** field specifies the system treatment of output:

OPOST	Postprocess output.
OLCUC	Map lowercase to uppercase on output.
ONLCR	Map NL to CR-NL on output.
OCRNL	Map CR to NL on output.
ONOCR	No CR output at column 0.
ONLRET	NL performs CR function.
OFILL	Use fill characters for delay.
OFDEL	Fill is DEL, else NUL.
NLDLY	Select new-line delays:
NL0	No delay
NL1	Delay type 1
CRDLY	Select carriage-return delays:
CR0	No delay
CR1	Delay type 1
CR2	Delay type 2
CR3	Delay type 3
TABDLY	Select horizontal-tab delays:
TAB0	No delay
TAB1	Delay type 1
TAB2	Delay type 2
TAB3	Expand tabs to spaces.
XTABS	Expand tabs to spaces.
BSDLY	Select backspace delays:
BS0	No delay
BS1	Delay type 1
VTDLY	Select vertical-tab delays:
VT0	No delay
VT1	Delay type 1
FFDLY	Select form-feed delays:
FF0	No delay
FF1	Delay type 1

If **OPOST** is set, output characters are post-processed as indicated by the remaining flags; otherwise characters are transmitted without change.

If **OLCUC** is set, a lowercase alphabetic character is transmitted as the corresponding uppercase character. This function is often used in conjunction with **IUCLC**.

If **ONLCR** is set, the NL character is transmitted as the CR-NL character pair. If **OCRNL** is set, the CR character is transmitted as the NL character. If **ONOCR** is set, no CR character is transmitted when at column 0 (first position). If **ONLRET** is set, the NL character is assumed to do the carriage-return function; the column pointer will be set to 0, and the delays specified for CR will be used. If **ONLRET** is clear, the NL character is assumed to perform only the line-feed function; the delays specified for NL are used and the column pointer remains unchanged. For all of these cases, the column pointer is always set to 0 if the CR character is actually transmitted.

The delay bits specify how long transmission stops to allow for mechanical or other movement when certain characters are sent to the terminal. The values of **NL0**, **CR0**, **TAB0**, **BS0**, **VT0**, and FF0 indicate no delay. If **OFILL** is set, fill characters are transmitted for delay instead of a timed delay. This is useful for high baud rate terminals, that need only a minimal delay. If **OFDEL** is set, the fill character is DEL; otherwise NUL.

If a form-feed or vertical-tab delay is specified, it lasts for about 2 seconds.

New-line delay lasts about 0.10 seconds. If **ONLRET** is set, carriage-return delays are used instead of the new-line delays. If **OFILL** is set, two fill characters are transmitted.

Carriage-return delay type 1 depends on the current column position; type 2 is about 0.10 seconds; type 3 about 0.15 seconds. If **OFILL** is set, delay type 1 transmits two fill characters; type 2, four fill characters.

Horizontal-tab delay type 1 is depends on the current column position. Type 2 is about 0.10 seconds; type 3 specifies that tabs are to be expanded into spaces. If **OFILL** is set, two fill characters are transmitted for any delay.

Backspace delay lasts about 0.05 seconds. If **OFILL** is set, one fill character is transmitted.

The actual delays depend on line speed and system load.

The initial output control value is all bits clear.

Control Modes

The **c_cflag** field describes the hardware control of the terminal:

CBAUD	**Baud rate:**	**CSIZE**	**Character size:**
B0	Hang up	CS5	5 bits
B50	50 baud	CS6	6 bits
B75	75 baud	CS7	7 bits
B110	110 baud	CS8	8 bits
B134	134.5 baud		
B150	150 baud	CSTOPB	Send two stop bits, else one.
B200	200 baud	CREAD	Enable receiver.
B300	300 baud	PARENB	Parity enable.
B600	600 baud	PARODD	Odd parity, else even.
B900	900 baud	HUPCL	Hang up on last close.
B1200	1200 baud	CLOCAL	Local line, else dial-up.
B1800	1800 baud	LOBLK	Reserved for use by *shl*(1).
B2400	2400 baud		
B3600	3600 baud		
B4800	4800 baud		
B7200	7200 baud		
B9600	9600 baud		
B19200	19200 baud		
B38400	38400 baud		
EXTA	External A		
EXTB	External B		

The CBAUD bits specify the baud rate. The zero baud rate, **B0**, is used to hang up the connection. If **B0** is specified, the modem control lines (see *modem*(7)) cease to be asserted. Normally, this disconnects the line. For any particular hardware, impossible speed changes are ignored. **CBAUD** is provided for use with the **termio** structure. When the **termios** structure is used, several routines are available for setting and getting the input and output baud rates (see *termios Structure Related Functions*).

The **CSIZE** bits specify the character size in bits for both transmission and reception. This size does not include the parity bit, if any. If **CSTOPB** is set, two stop bits are used; otherwise one stop bit. For example, at 110 baud, many devices require two stop bits.

If **PARENB** is set, parity generation is enabled (a parity bit is added to each output character). Furthermore, parity detection is enabled (incoming characters are checked for the correct parity). If **PARENB** is set, **PARODD** specifies odd parity if set; otherwise even parity is used. If **PARENB** is clear, both parity generation and parity checking are disabled.

If **CREAD** is set, the receiver is enabled. Otherwise no characters can be received.

The specific effects of the **HUPCL** and **CLOCAL** bits depend on the mode and type of the modem control in effect. See *modem*(7) for the details.

If **HUPCL** is set, the modem control lines for the port are lowered (disconnected) when the last process using the open port closes it or terminates.

If **CLOCAL** is set, a connection does not depend on the state of the modem status lines. If **CLOCAL** is clear, the modem status lines are monitored.

Under normal circumstances, a call to **read()** waits for a modem connection to complete. However, if either the **O_NDELAY** or the **O_NONBLOCK** flags are set or **CLOCAL** is set, the **open()** returns immediately without waiting for the connection. If **CLOCAL** is set, see *Modem Disconnect* for the effects of **read()** and **write()** for those files for which the connection has not been established or has been lost.

LOBLK is used by the shell layers facility (see *shl*(1)). The shell layers facility is not part of the general terminal interface, and the **LOBLK** bit is not examined by the general terminal interface.

The initial hardware control value after open is **B300**, **CS8**, **CREAD**, and **HUPCL**.

Local Modes

The **c_lflag** field is used to control terminal functions.

ISIG	Enable signals.
ICANON	Canonical input (erase and kill processing).
XCASE	Canonical upper/lower presentation.
ECHO	Enable echo.
ECHOE	Echo ERASE as correcting backspace sequence.
ECHOK	Echo NL after kill character.
ECHONL	Echo NL.
NOFLSH	Disable flush after interrupt, quit, or suspend.
TOSTOP	Send SIGTTOU for background output.
ECHOCTL	Echo control characters as ^char, DEL as ^?.
ECHOPRT	Echo erased character as character is erased.
ECHOKE	BS SP BS erase entire line on line kill.
FLUSHO	Output is being flushed.
PENDIN	Reprocess pending input at next read or input character.
IEXTEN	Enable extended functions.

If **ISIG** is set, each input character is checked against the special control characters INTR, QUIT, SUSP, and DSUSP (see *Process Group Control IOCTL Commands*). If an input character matches one of these control characters, the function associated with that character is performed and the character is discarded. If **ISIG** is clear, no checking is done and the character is treated as a normal data character. Thus these special input functions are possible only if **ISIG** is set.

If **ICANON** is set, canonical processing is enabled. This enables the erase and kill edit functions, and the assembly of input characters into lines delimited by NL, EOF, EOL, or EOL2. If **ICANON** is clear, read requests are satisfied directly from the input queue. A read blocks until at least MIN characters have been received or the timeout value TIME has expired between characters. (See *Non-Canonical Mode Input Processing (MIN/TIME Interaction)*). This allows fast bursts of input to be read efficiently while still allowing single-character input. The time value represents tenths of seconds.

If **XCASE** is set, and if **ICANON** is set, an uppercase letter is accepted on input by preceding it with a **** character, and is output preceded by a **** character. In this mode, the following escape sequences are generated on output and accepted on input:

```
To obtain:   Use:
    `         \'
    |         \!
    {         \(
    }         \)
    \         \\
```

For example, **A** is input as **\a**, **\n** as **\\\n**, and **\N** as **\\\\\n**. **XCASE** would normally be used in conjunction with **IUCLC** and **OLCUC** for terminals that support only the first-sixty-four-character limited character set. In this case, **IUCLC** processing is done before **XCASE** for input, and processing is done after **XCASE** for output. Therefore typing **A** causes an **a** to be read because of **IUCLC**, and typing **\A** causes an **A** to be read since **IUCLC** produces **\a** which is turned into **A** by the **XCASE** processing.

If **ECHO** is set, characters are echoed back to the terminal when received. If **ECHO** is clear, characters are not echoed.

When **ICANON** is set, canonical processing is enabled. This enables the erase and kill edit functions, and the assembly of input characters into lines delimited by NL, EOF, EOL and EOL2 as described in *Canonical Mode Input Processing*. Furthermore, the following echo functions are possible.

If **ECHO** and **ECHOE** are set, the ERASE and WERASE characters are echoed as the three-character ASCII sequence BS SP BS, which clears the last character or word from the CRT screen.

If **ECHO** and **ECHOPRT** are set, and **ECHOE** is clear, the first ERASE and WERASE character in a sequence echoes a backslash (\) followed by the characters being erased. Subsequent ERASE or WERASE characters echo the characters being erased in reverse order. The next non-erase character causes a slash (/) to be typed before it is echoed.

If **ECHOKE** and **ECHO** are set, the KILL character is echoed by erasing each character on the line from the CRT screen using using the method selected by **ECHOE** and **ECHOPRT.**

If **ECHOCTL** and **ECHO** are set, all control characters (characters with codes between 0 and 37 octal) other than ASCII TAB, ASCII NL, the START and STOP characters, ASCII CR, and ASCII BS are echoed as ˆchar, where char is the character given by adding 100 octal to the control character's code.

If **ECHOK** is set and **ECHOKE** is not set, the NL character is echoed after the kill character to emphasize that the line is being deleted.

If **ECHONL** is set, the NL character is echoed even if **ECHO** is clear. This is useful for terminals set to local echo (that is, half duplex).

Unless escaped, the EOF character is not echoed. Because ASCII EOT is the default EOF character, this prevents terminals that respond to EOT from hanging up.

If **NOFLSH** is set, the normal flush of the input and output queues associated with quit, interrupt, and suspend characters is not done. However, **NOFLSH** does not affect the flushing of data upon receipt of a break when **BRKINT** is set.

If the **TOSTOP** bit is set, an attempt by a process that is not in the foreground process group to write to its controlling terminal will be denied when the process is not ignoring and not blocking the **SIGTTOU** signal. If the write is denied and the process is a member of an orphaned process group **write()** returns –1 and sets **errno** to **EIO** and no signal is sent. If the write is denied and the process is a not a member of an orphaned process group, the **SIGTTOU** signal is sent to that process group.

If **FLUSHO** is set, data written to the terminal device is discarded. This bit is set by a program. A program can cancel the **FLUSHO** effect by clearing **FLUSHO**.

If **PENDIN** is set, any input that has not been read is reprocessed and possibly re-echoed when the next character arrives as input.

If **ICANON** is set, the ERASE, KILL, and EOF characters can be escaped by a preceding **** character, in which case no special function is done.

IEXTEN must be set before the **ECHOCTL**, **ECHOPRT**, **ECHOKE**, **FLUSHO**, and **PENDIN** functions are allowed. In addition, the special characters WERASE and LNEXT are allowed only if **IEXTEN** is set. **IEXTEN** does not affect any other functions.

The initial local control value is all-bits-clear.

Special Control Characters

Special control characters are defined in the array **c_cc**. All of these special characters can be changed. The subscript name and description for each element in both canonical and non-canonical mode are shown in the following table.

Subscript Usage Canonical	Subscript Usage Non-Canonical	Description
VEOF		EOF character
VEOL		EOL character
VEOL2		EOL2 character
VERASE		ERASE character
VWERASE		WERASE character
VINTR	VINTR	INTR character
VKILL		KILL character
	VMIN	MIN value
VQUIT	VQUIT	QUIT character
VSTART	VSTART	START character
VSTOP	VSTOP	STOP character
VSUSP	VSUSP	SUSP character
VDSUSP	VDSUSP	DSUSP character
	VTIME	TIME value
VLNEXT	VLNEXT	LNEXT character

t

termios Structure-Related Functions

The following functions are provided when using the *termios* structure. Note that the effects on the terminal device of the **cfsetispeed()** and **cfsetospeed()** functions do not become effective until the **tcsetattr()** function is successfully called. Refer to the appropriate manual entries for details.

termios Structure Functions

Function	Description
`cfgetospeed()`	get output baud rate
`cfgetispeed()`	get input baud rate
`cfsetospeed()`	set output baud rate
`cfsetispeed()`	set input baud rate
`tcgetattr()`	get terminal state
`tcsetattr()`	set terminal state

termio Structure-Related OCTL Commands

Several **ioctl()** system calls apply to terminal files that use the **termio** structure (see *termio Structure*). If a requested command is not recognized, the request returns –1 with **errno** set to [EINVAL].

ioctl() system calls that reference the **termio** structure have the form:

```
ioctl (fildes, command, arg)
struct termio *arg;
```

Commands using this form are:

TCGETA Get the parameters associated with the terminal and store them in the **termio** structure referenced by *arg*. This command is allowed from a background process; however, the information may be subsequently changed by a foreground process.

TCSETA Set the parameters associated with the terminal from the **termio** structure referenced by *arg*. The change is immediate. If characters are being output when the command is requested, results are undefined and the output may be garbled.

TCSETAW Wait for the output to drain before setting new parameters. This form should be used when changing parameters that affect output.

TCSETAF Wait for the output to drain, then flush the input queue and set the new parameters.

termio Caveats

Only the first eight special control characters (see *termios Structure*) can be set or returned. The values of indices VEOL and VEOF are the same as indices VTIME and VMIN respectively. Hence if **ICANON** is set, VEOL or VTIME is the additional end-of-line character and VEOF or VMIN is the end-of-file character. If **ICANON** is clear, VEOL or VTIME is the inter-character-timer value and VEOF or VMIN is the minimum number of characters desired for reads.

Structure-Independent Functions

The following functions which are independent of both the **termio** and **termios** structures are provided for controlling terminals. Refer to the appropriate manual entries for details.

Structure-Independent Functions

Function	Description
`tcsendbreak()`	send a break
`tcdrain()`	wait until output has drained
`tcflush()`	flush input or output queue or both
`tcflow()`	suspend or resume input or output
`tcgetpgrp()`	get foreground process group id
`tcsetpgrp()`	set foreground process group id
`tcgetsid()`	get session id

t

System Asynchronous I/O IOCTL Commands

The following **ioctl()** system calls provide for system asynchronous I/O and have the form:

```
ioctl (fildes, command, arg)
int *arg;
```

Commands using this form are:

FIOSSAIOSTAT If the integer referenced by *arg* is non-zero, system asynchronous I/O is enabled; that is, enable **SIGIO** to be sent to the process currently designated with **FIOSSAIOOWN** (see below) whenever the terminal device file status changes from "no read data available" to "read data available". If no process has been designated with **FIOSSAIOOWN**, enable **SIGIO** to be sent to the first process that opened the terminal device file.

If the designated process has exited, the **SIGIO** signal is not sent to any process.

If the integer referenced by *arg* is 0, system asynchronous I/O is disabled.

The default on open of a terminal device file is that system asynchronous I/O is disabled.

FIOGSAIOSTAT The integer referenced by *arg* is set to 1 if system asynchronous I/O is enabled. Otherwise, the integer referenced by *arg* is set to 0.

FIOSSAIOOWN Set the process ID that will receive the **SIGIO** signals due to system asynchronous I/O to the value of the integer referenced by *arg*. If no process can be found corresponding to that specified by the integer referenced by *arg*, the call returns –1 with **errno** set to [ESRCH]. A user with appropriate privileges can designate that any process receive the **SIGIO** signals. If the request is not made by a user with appropriate privileges and the calling process does not either designate that itself or another process whose real, saved, or effective user ID matches its real or effective user ID or the calling process does not designate a process that is a descendant of the calling process to receive the **SIGIO** signals, the call returns –1 with **errno** set to [EPERM]. See *privileges*(5) for more information about privileged access on systems that support fine-grained privileges.

If the designated process subsequently exits, the **SIGIO** signal is not sent to any process.

The default on open of a terminal device file is that the process performing the first open is set to receive the **SIGIO** signals.

FIOGSAIOOWN The integer referenced by *arg* is set to the process ID designated to receive **SIGIO** signals.

Line Control IOCTL Commands

Several **ioctl()** system calls control input and output. Some of these calls have the form:

```
ioctl (fildes, command, arg)
int arg;
```

Commands using this form are:

TCSBRK Wait for the output to drain. If *arg* is 0, send a break (zero bits for at least 0.25 seconds). The **tcsendbreak()** function performs the same function (see *tcsendbreak*(3C)).

TCXONC Start/stop control. If *arg* is 0, suspend output; if 1, restart suspended output; if 2, transmit a STOP character; if 3, transmit a START character. If any other value is given for *arg,* the call returns –1 with **errno** set to [EINVAL]. The **tcflow()** function performs the same functions (see *tcflow*(3C)).

t

TCFLSH If *arg* is 0, flush the input queue; if 1, flush the output queue; if 2, flush both the input and output queues. If any other value is given for *arg,* the call returns –1 with **errno** set to [EINVAL]. The **tcflush()** function performs the same functions (see *tcflush*(3C)).

Sending a BREAK is accomplished by holding the data transmit line at a SPACE or logical zero condition for at least 0.25 seconds. During this interval, data can be sent to the device, but because of serial data interface limitations, the BREAK takes precedence over all data. Thus, all data sent to a device during a BREAK is lost. This includes system-generated XON/XOFF characters used for input flow control. Note also that a delay in transmission of the XOFF flow control character until after the BREAK is terminated could still result in data overflow because the flow control character may not be sent soon enough.

Other calls have the form:

```
ioctl (fildes, command, arg)
int *arg;
```

Commands using this form are:

FIONREAD Returns in the integer referenced by *arg* the number of characters immediately readable from the terminal device file. This command is allowed from a background process; however, the data itself cannot be read from a background process.

Non-blocking I/O IOCTL Commands

Non-blocking I/O is easily provided via the **O_NONBLOCK** and **O_NDELAY** flags available in both *open*(2) and *fcntl*(2). The commands in this section are provided for backward compatibility with previously developed applications. **ioctl()** system calls that provide a style of non-blocking I/O different from **O_NONBLOCK** and **O_NDELAY** have the form:

```
ioctl (fildes, command, arg)
int *arg;
```

Commands using this form are:

FIOSNBIO If the integer referenced by *arg* is non-zero, **FIOSNBIO**-style non-blocking I/O is enabled; that is, subsequent reads and writes to the terminal device file are handled in a non-blocking manner (see below). If the integer referenced by *arg* is 0, **FIOSNBIO**-style non-blocking I/O is disabled.

For reads, **FIOSNBIO**-style non-blocking I/O prevents all read requests to that device file from blocking, whether the requests succeed or fail. Such a read request completes in one of three ways:

- If there is enough data available to satisfy the entire request, the read completes successfully, having read all of the data, and returns the number of characters read;
- If there is not enough data available to satisfy the entire request, the read completes successfully, having read as much data as possible, and returns the number of characters read;
- If there is no data available, the read returns –1 with **errno** set to [EWOULDBLOCK].

For writes, **FIOSNBIO**-style non-blocking I/O prevents all write requests to that device file from blocking, whether the requests succeed or fail. Such a write request completes in one of three ways:

- If there is enough space available in the system to buffer all the data, the write completes successfully, having written out all of the data, and returns the number of characters written;
- If there is not enough space in the buffer to write out the entire request, the write completes successfully, having written as much data as possible, and returns the number of characters written;
- If there is no space in the buffer, the write returns –1 with **errno** set to [EWOULDBLOCK].

To prohibit **FIOSNBIO**-style non-blocking I/O from interfering with the **O_NONBLOCK** and **O_NDELAY** flags (see *open*(2) and *fcntl*(2)), the functionality of **O_NONBLOCK** and **O_NDELAY** always supersedes the functionality of **FIOSNBIO**-style non-blocking I/O. This means that if either **O_NONBLOCK** or **O_NDELAY** is set, the driver performs read requests in accordance with the definition of **O_NDELAY** or **O_NONBLOCK.** When both **O_NONBLOCK** and **O_NDELAY** are clear, the definition of **FIOSNBIO**-style non-blocking I/O applies.

The default on open of a terminal device file is that **FIOSNBIO**-style non-blocking I/O is disabled.

FIOGSNBIO The integer referenced by *arg* is set to 1, if **FIOSNBIO**-style non-blocking I/O is enabled. Otherwise, the integer referenced by *arg* is set to 0.

Process Group Control IOCTL Commands

The process group control features described here (except for setting and getting the delayed stop process character) are easily implemented using the functions **tcgetattr()**, **tcsetattr()**, **tcgetpgrp()**, **tcsetpgrp()**, and **tcsetsid()**, (see *tcattribute*(3C), *tcgetpgrp*(3C), *tcsetpgrp*(3C), and *tcgetsid*(3C) respectively).

The following structure, used with process group control, is defined in **<bsdtty.h>**:

```
struct ltchars  {
      unsigned char t_suspc;   /* stop process character*/
      unsigned char t_dsuspc;  /* delayed stop process character*/
      unsigned char t_rprntc;  /* reserved; must be '_POSIX_VDISABLE'*/
      unsigned char t_flushc;  /* reserved; must be '_POSIX_VDISABLE'*/
      unsigned char t_werasc;  /* reserved; must be '_POSIX_VDISABLE'*/
      unsigned char t_lnextc;  /* reserved; must be '_POSIX_VDISABLE'*/
};
```

The initial value for all these characters is **_POSIX_VDISABLE**, which causes them to be disabled. The meaning for each character is as follows:

t_suspc Suspend the foreground process group. A *suspend* signal (**SIGTSTP**) is sent to all processes in the foreground process group. Normally, each process is forced to stop, but arrangements can be made to either ignore or block the signal, or to receive a trap to an agreed-upon location; see *signal*(2) and *signal*(5). When enabled, the typical value for this character is Control-Z or ASCII SUB. Setting or getting **t_suspc** is equivalent to setting or getting the SUSP special control character.

t_dsuspc Same as **t_suspc**, except that the *suspend* signal (**SIGTSTP**) is sent when a process reads the character, rather than when the character is typed. When enabled, the typical value for this character is Control-Y or ASCII EM.

Attempts to set any of the reserved characters to a value other than **_POSIX_VDISABLE** cause **ioctl()** to return –1 with **errno** set to [EINVAL] with no change in value of the reserved character.

ioctl() system calls that use the above structure have the form:

```
ioctl (fildes, command, arg)
struct ltchars *arg;
```

Commands using this form are:

TIOCGLTC Get the process group control characters and store them in the *ltchars* structure referenced by *arg*. This command is allowed from a background process. However, the information may be subsequently changed by a foreground process.

TIOCSLTC Set the process group control characters from the structure referenced by *arg*.

Additional process group control **ioctl()** system calls have the form:

```
ioctl (fildes, command, arg)
unsigned int *arg;
```

Commands using this form are:

TIOCGPGRP Returns in the integer referenced by *arg* the foreground process group associated with the terminal. This command is allowed from a background process. However, the information may be subsequently changed by a foreground process. This feature is easily implemented using the **tcgetpgrp()** function (see *tcgetpgrp*(3C)).

If the **ioctl()** call fails, it returns –1 and sets **errno** to one of the following values:

[EBADF] *fildes* is not a valid file descriptor.

[ENOTTY] The file associated with *fildes* is not the controlling terminal, or the calling process does not have a controlling terminal.

[EACCES] The file associated with *fildes* is the controlling terminal of the calling process, however, there is no foreground process group defined for the controlling terminal.

Note: [EACCES] may not be returned in future releases. Behavior in cases where no foreground process group is defined for the controlling terminal may change in future versions of the POSIX standard. Portable applications, therefore, should not rely on this error condition.

TIOCSPGRP Sets the foreground process group associated with the terminal to the value referenced by *arg*. This feature is easily implemented using the **tcsetpgrp()** function (see *tcsetpgrp*(3C)).

If the **ioctl()** call fails, it returns –1 and sets **errno** to one of the following values:

[EBADF] *fildes* is not a valid file descriptor.

[EINVAL] The process ID referenced by *arg* is not a supported value.

[ENOTTY] The calling process does not have a controlling terminal, or the *fildes* is not the controlling terminal, or the controlling terminal is no longer associated with the session of the calling process.

[EPERM] The process ID referenced by *arg* is a supported value but does not match the process group ID of a process in the same session as the calling process.

TIOCGSID Returns in the integer referenced by *arg* the session ID of the terminal specified by fildes. This feature is easily implemented using the **tcgetsid()** function (see *tcgetsid*(3C)).

If the **ioctl()** call fails, it returns –1 and sets **errno** to one of the following values:

[EBADF] *fildes* is not a valid file descriptor.

[ENOTTY] The device associated with *fildes* is not a terminal.

[EACCES] The *fildes* is a terminal that is not allocated to a session.

TIOCLGET Get the process group control mode word and store it in the int referenced by *arg*. This command is allowed from a background process; however, the information may be subsequently changed by a foreground process.

TIOCLSET Set the process group control mode word to the value of the int referenced by *arg*.

TIOCLBIS Use the int referenced by *arg* as a mask of bits to set in the process group control mode word.

TIOCLBIC Use the int referenced by *arg* as a mask of bits to clear in the process group control mode word.

The following bit is defined in the process group control mode word:

LTOSTOP Send **SIGTTOU** for background writes.

Setting or clearing **LTOSTOP** is equivalent to setting or clearing the **TOSTOP** flag (see *Local Modes*). If **LTOSTOP** is set and a process is not in the foreground process group of its controlling terminal, a write by the process to its controlling terminal may be denied (see *Terminal Access Control*).

Terminal Size IOCTL Commands

The following **ioctl()** system calls are used to get and set terminal size information for the terminal referenced by *fildes*. These **ioctl()** system calls use the **winsize** structure to get and set the terminal size information. The **winsize** structure, defined in **<termios.h>**, has the following members :

```
unsigned short  ws_row;     /* Rows, in characters */
unsigned short  ws_col;     /* Columns, in characters */
unsigned short  ws_xpixel;  /* Horizontal size, in pixels */
unsigned short  ws_ypixel;  /* Vertical size, in pixels */
```

The initial values for all elements of terminal size are zero. The values for terminal size are neither set nor used by the general terminal interface, and have no effect on the functionality of the general terminal interface. The values for terminal size are set and used only by applications that access them through the terminal-size **ioctl()** system calls (see *ioctl*(2)).

ioctl() system calls that use the above structure have the form:

```
ioctl (fildes, command, arg)
struct winsize *arg;
```

t

Commands using this form are:

TIOCGWINSZ Get the terminal size values and store them in the **winsize** structure referenced by *arg*. This command is allowed from a background process.

TIOCSWINSZ Set the terminal size values from the **winsize** structure referenced by *arg*. If any of the new values differ from previous values, a **SIGWINCH** signal is sent to all processes in the terminal's foreground process group.

Console Output Redirection IOCTL Command

Output which would normally be sent to the system console may be redirected to any other TTY device or pseudo-device in the system. The **ioctl()** system call used to control console output redirection has the form:

```
ioctl (fildes, command, arg)
int arg;
```

The command using this form is:

TIOCCONS Redirect system console output. Any output that would normally be sent to the system console, either through kernel printf requests, or through the console special file, will instead be sent to the terminal referenced by *fildes*. The value of *arg* is ignored. The user must have the **DEVOPS** privilege to execute this request. Otherwise, the call returns −1 with **errno** set to [EPERM]. If the console output has not been redirected to a different device by a later call to this command, it is redirected back to the physical console device when *fildes* is closed.

WARNINGS

Various HP-UX implementations use non-serial interfaces that look like terminals (such as bit-mapped graphics displays) or "smart cards" that cannot implement the exact capabilities described above. Therefore, not all systems can exactly meet the standard stated above. Each implementation is required to state any deviations from the standard as part of its system-specific documentation.

FIOSSAIOSTAT is similar to BSD 4.2 **FIOASYNC**, with the addition of provisions for security.

FIOGSAIOSTAT is of HP origin, complements **FIOSSAIOSTAT**, and allows saving and restoring system asynchronous I/O TTY states for command interpreter processes.

FIOSSAIOOWN is similar to BSD 4.2 **FIOSETOWN**, with additional provisions for security.

FIOGSAIOOWN is similar to BSD **FIOGETOWN**. 4.2 Note also the difference that the BSD 4.2 version of this functionality used process groups, while the HP-UX version only uses processes.

FIOSNBIO is the same as BSD **FIONBIO**, 4.2 except that it does not interfere with the **O_NDELAY** or **O_NONBLOCK open()** and **fcntl()** flags.

FIOGNBIO is of HP origin, complements **FIOSNBIO**, and allows saving and restoring the **FIOSNBIO**-style non-blocking I/O TTY state for command interpreter processes.

The general terminal interface uses a system resource known as a **cblock** to store data being transmitted or received through a communications port. These cblocks are continuously used and freed for reuse as data pass through the system. If too few cblocks are configured in the system, the cblock pool may be temporarily or permanently exhausted, and data loss, system hangs, or reduced system performance can result.

If cblock exhaustion is suspected, you can examine the system message buffer with **dmesg** (see *dmesg*(1M)) for messages indicating cblock exhaustion has occurred. Or, you can use **adb** (see *adb*(1)) if examining the corefile of a dump. The message format is

```
WARNING: cblock exhaustion occurred n times
```

where n indicates the number of times the operating system has requested a cblock and none could be provided. If this message is observed, the kernel should be reconfigured to generate a larger number of cblocks.

A cblock is 32 bytes in length. The default number of cblocks configured in the system is defined to be 8292.

This can be overridden by using the optional tunable system parameter **nclist** to specify the desired number of cblocks to be used in the system.

SAM or *kctune*(1M) may be used to change the **nclist** value.

DEPENDENCIES

Workstations

Built-in serial ports on workstation machines support the following additional baud rate settings: 57600, and 115200. An RS-232-to-RS-422 converter may be required to achieve practical cable lengths at these baud rates (because RS-232 only specifies up to 19200 baud).

Timed delays are not supported.

Built-in serial ports on workstation systems have RTS and CTS flow control capability, configurable receive FIFO trigger levels, and a configurable transmit limit. RTS/CTS hardware handshaking can be enabled through a bit in the device file minor number, through an **ioctl()** call (see *termiox*(7)), or through the **stty** command (see *stty*(1)).

The receive FIFO trigger level is configurable through two bits in the device file minor number. The receive FIFO trigger level is used to set the level at which a receive interrupt is generated to the system. Setting a smaller value for the receive FIFO trigger level enables the system to react more quickly to receipt of characters. However, using a smaller trigger level increases system overhead to process the additional interrupts. A higher receive FIFO trigger level reduces the system interrupt overhead for heavy inbound data traffic at the cost of less time for the system to read data from the hardware before receive FIFOs are overrun. When using RTS flow control, the receive FIFO trigger level also determines the point at which the hardware lowers RTS to protect the receive FIFO. Use of a higher receive FIFO trigger level also reduces XOFF flow control responsiveness because, under light inbound data flow conditions, receipt of the XOFF character by the system is slightly delayed. Choice of the appropriate receive FIFO trigger level should be based upon how the serial port is to be used. For most applications a receive FIFO trigger level of 8 (c3,c2 = 10) is suggested.

Two bits in the device file minor number specify the transmit limit, the number of characters which are successively loaded into the transmit FIFO. Setting a smaller transmit limit allows the transmitter to be more responsive to flow control either from receipt of an XOFF character or de-assertion of CTS at the cost of increased system interrupt overhead. Setting a larger transmit limit reduces interrupt overhead but is not as responsive to flow control since the remainder of the transmit FIFO can be transmitted even after the transmitter is flow controlled. When communicating with devices which have little tolerance for data receipt after flow control, one must choose the transmit limit appropriately.

Device File Minor Number

Workstation device file minor numbers take the form:

0x*IIC***0***HM*

where:

II = Two hexadecimal digits (8 bits) to indicate the instance of the serial interface.

C = One hexadecimal digit (4 bits) for FIFO control. Values for each bit are as follows:

Receive FIFO Trigger Level			Transmit Limit		
c3	**c2**	**Level**	**c1**	**c0**	**Limit**
0	0	1	0	0	1
0	1	4	0	1	4
1	0	8	1	0	8
1	1	14	1	1	12

H = One hexadecimal digit (4 bits) which controls diagnostic access and hardware flow control.

Bit	Value
h3	Diagnostic telephony access
h2	Reserved
h1	Reserved
h0	Enables RTS/CTS hardware flow control

M = One hexadecimal digit (4 bits) to determine the port access type. Values for each bit are as follows:

Bit	Value
m3	TI/ALP
m2	0 = Simple protocol (U.S.), 1 = CCITT protocol (Europe)
m1m0	00 = Direct 01 = Dial-out modem 10 = Dial-in modem 11 = Invalid

Servers

Timed output delays are not directly supported. If used, an appropriate number of fill characters (based on the current baud rate) is output. The total time to output the fill characters is at least as long as the time requested.

The system specified input flow control values are as follows: low water mark is 60, high water mark is 180, and maximum allowed input is 512.

The HP 98196A (formerly 27140A option 800) interface does not support the following hardware settings:

CBAUD **B200**, **B38400**, **EXTA**, **EXTB**.

The HP A1703-60003 and the HP 28639-60001 interfaces do not support baud rates above 9600. Furthermore, changing the following hardware settings on port 0 from the default (9600 baud, 8 bit characters, 1 stop bit, no parity) is not supported:

CBAUD, **CSIZE**, **CSTOPB**, **PARENB**, **PARODD**.

The HP J2094A interface does not support baud rates above 19200.

The HP J2094A supports RTS and CTS flow control. The RTS/CTS hardware handshaking can be enabled through a bit in the device file minor number, through an **ioctl()** call (see *termiox*(7)), or through the **stty** command (see *stty*(1)).

Device File Minor Number

Server device file minor numbers take the form:

0x*IIPPHM*

where:

II = Two hexadecimal digits (8 bits) to indicate the instance of the serial interface.

PP = Two hexadecimal digits (8 bits) to indicate the port number of this device on the serial interface.

H = One hexadecimal digit (4 bits) which controls diagnostic access and hardware flow control (HP J2094A only).

Bit	Value
h3	Card diagnostic
h2	Port diagnostic
h1	Reserved
h0	Enables RTS/CTS hardware flow control

M = One hexadecimal digit (4 bits) for the port access type. Values for each bit are as follows:

t

Bit	Value
m3	TI/ALP
m2	0 = Simple protocol (U.S.), 1 = CCITT protocol (Europe)
m1m0	00 = Direct 01 = Dial-out modem 10 = Dial-in modem 11 = Invalid

AUTHOR

termios was developed by HP and the IEEE Computer Society.

termio was developed by HP, AT&T, and the University of California, Berkeley.

FILES

/dev/console
/dev/cua*
/dev/cul*
/dev/tty*
/dev/ttyd*

SEE ALSO

adb(1), shl(1), stty(1), dmesg(1M), kctune(1M), mknod(1M), fork(2), ioctl(2), setpgid(2), setsid(2), signal(2), stty(2), cfspeed(3C), tcattribute(3C), tccontrol(3C), tcgetpgrp(3C), tcgetsid(3C), tcsetpgrp(3C), privileges(5), signal(5), unistd(5), modem(7), sttyV6(7), termiox(7), tty(7).

STANDARDS CONFORMANCE

termio: SVID2, SVID3, XPG2

termios: AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1

t

NAME

termiox - extended general terminal interface

SYNOPSIS

#include <sys/termiox.h>

ioctl (int *fildes*, **int** *request*, **struct termiox *** *arg)*

DESCRIPTION

The extended general terminal interface supplements the *termio*(7) general terminal interface by adding support for asynchronous hardware flow control and local implementations of additional asynchronous features. Some systems may not support all of these capabilities because of hardware or software limitations. Other systems may not permit certain functions to be disabled. In such cases, the appropriate bits are ignored. If the capabilities can be supported, the interface described here must be used.

Hardware Flow Control Modes

Hardware flow control supplements the termio **IXON**, **IXOFF**, and **IXANY** character flow control (see *termio*(7)). Character flow control occurs when one device controls the data transfer of another device by inserting control characters in the data stream between devices. Hardware flow control occurs when one device controls the data transfer of another device by using electrical control signals on wires (circuits) of the asynchronous interface. Character flow control and hardware flow control can be simultaneously set.

In asynchronous, full duplex applications, the use of the Electronics Industries Association's EIA-232-D Request To Send (RTS) and Clear To Send (CTS) circuits is the preferred method of hardware flow control.

The EIA-232-D standard specified only unidirectional hardware flow control where the Data Circuit-terminating Equipment or Data Communications Equipment (DCE) indicates to the Data Terminal Equipment (DTE) to stop transmitting data. The termiox interface allows both unidirectional and bidirectional hardware flow control; when bidirectional flow control is enabled, either the DCE or DTE can indicate to each other to stop transmitting data across the interface.

Clock Modes

Isochronous flow control and clock mode communication are not supported.

Terminal Parameters

Parameters that control the behavior of devices providing the termiox interface are specified by the **termiox** structure, defined in the **<sys/termiox.h>** header file. Several **ioctl()** system calls (see *ioctl*(5)) that fetch or change these parameters use the **termiox** structure which contains the following members:

```
unsigned short x_hflag;     /* hardware flow control modes */
unsigned short x_cflag;     /* clock modes */
unsigned short x_rflag;     /* reserved modes */
unsigned short x_sflag;     /* spare local modes */
```

The **x_hflag** field describes hardware flow control modes:

RTSXOFF	**0000001**	Enable RTS hardware flow control on input.
CTSXON	**0000002**	Enable CTS hardware flow control on input.

The RTS and CTS circuits are involved in establishing CCITT modem connections. Since RTS and CTS circuits are used both by CCITT modem connections and by hardware flow control, CCITT modem and hardware flow control cannot be simultaneously enabled.

Variations of different hardware flow control methods can be selected by setting the appropriate bits. For example, bidirectional RTS/CTS flow control is selected by setting both the **RTSXOFF** and **CTSXON** bits. Unidirectional CTS hardware flow control is selected by setting only the **CTSXON** bit.

If **RTSXOFF** is set, the Request to Send (RTS) circuit (line) is raised, and if the asynchronous port needs to have its input stopped, it lowers the Request to Send (RTS) line. If the RTS line is lowered, it is assumed that the connected device will stop its output until RTS is raised.

If **CTSXON** is set, output occurs only if the Clear To Send (CTS) circuit (line) is raised by the connected device. If the CTS line is lowered by the connected device, output is suspended until CTS is raised.

termiox Structure Related IOCTL Command

The **ioctl()** system calls that reference the **termiox** structure have the form:

```
ioctl (fildes, command, arg)
struct termiox *arg;
```

Commands using this form are:

TCGETX The argument is a pointer to a **termiox** structure. The current terminal parameters are fetched and stored into that structure.

TCSETX The argument is a pointer to a **termiox** structure. The current terminal parameters are set from the values stored in that structure. The change is immediate. Errors that can be returned include:

[EINVAL] The port does not support hardware flow control.

[ENOTTY] The file descriptor for this port is configured for CCITT mode access. Hardware flow control is not allowed on CCITT mode devices.

TCSETXW The argument is a pointer to a **termiox** structure. The current terminal parameters are set from the values stored in that structure. The change occurs after all characters queued for output have been transmitted. This form should be used when changing parameters that affect output. Errors that can be returned include:

[EINVAL] The port does not support hardware flow control.

[ENOTTY] The file descriptor for this port is configured for CCITT mode access. Hardware flow control is not allowed on CCITT mode devices.

TCSETXF The argument is a pointer to a **termiox** structure. The current terminal parameters are set from the values stored in that structure. The change occurs after all characters queued for output have been transmitted; all characters queued for input are discarded, then the change occurs. Errors that can be returned include:

[EINVAL] The port does not support hardware flow control.

[ENOTTY] The file descriptor for this port is configured for CCITT mode access. Hardware flow control is not allowed on CCITT mode devices.

AUTHOR

termiox was developed by HP and AT&T.

FILES

Files in or under **/dev/tty***.

SEE ALSO

ioctl(2), termio(7), modem(7).

NAME

timod - STREAMS module for converting ioctl() calls into Transport Interface messages

DESCRIPTION

The **timod** module is a STREAMS module that converts **ioctl()** calls from a transport user supporting the Transport Interface (TI) into messages that a transport protocol provider supporting TI can consume. This allows the user to initiate certain TI functions as atomic operations. This release of HP-UX no longer automatically pushes **timod** whenever a *t_open*(3) is performed. The TLI and XTI libraries have been modified to no longer require this module to perform the atomic operations described within this man page. Binary compatibility is not a problem since the module will still exist within the kernel. But, any application which is recompiled and expects the module to be automatically pushed, may not work without code modification.

The user places and removes the **timod** module on a device stream by calling the STREAMS **I_PUSH ioctl()** and **I_POP ioctl()** functions. (The TLI function **t_open()** pushes **timod** onto the device stream for the user.) The **timod** module should only be pushed onto streams which are terminated by transport providers which conform to the Transport Interface. *tirdwr*(7) is an alternative interface to **timod** which supports the **read()** and **write()** system calls. If **tirdwr** has been pushed onto the stream, the user should use the **I_POP ioctl** to remove the **tirdwr** module from the stream before pushing **timod**.

The **timod** module transparently passes any STREAMS messages that are not generated by the **ioctl()** commands described below to the neighboring module or driver. **timod** will act on an **I_STR ioctl()** whose **strioctl.ic_cmd** field is one of the values below. (See *streamio*(7) for a description of the **I_STR ioctl** and the **strioctl** structure.)

TI_BIND This TI command binds an address to the transport protocol provider. The STREAMS message that the module issues to the **TI_BIND ioctl()** call is equivalent to the TI message type **T_bind_req**. The STREAMS message that the module returns in response to the successful completion of the **TI_BIND ioctl()** call is equivalent to the TI message type **T_bind_ack**.

TI_UNBIND This TI command unbinds an address from the transport protocol provider. The STREAMS message that the module issues to the **TI_UNBIND ioctl()** call is equivalent to the TI message type **T_unbind_req**. The STREAMS message that the module returns in response to the successful completion of the **TI_UNBIND ioctl()** call is equivalent to the TI message type **T_ok_ack**.

TI_GETINFO This TI command gets the TI protocol-specific information from the transport protocol provider. The STREAMS message that the module issues to the **TI_GETINFO ioctl()** call is equivalent to the TI message type **T_info_req**. The STREAMS message that the module returns in response to the successful completion of the **TI_GETINFO ioctl()** call is equivalent to the TI message type **T_info_ack**.

TI_OPTMGMT This TI command gets, sets, or negotiates TI protocol-specific options with the transport protocol provider. The STREAMS message that the module issues to the **TI_OPTMGMT ioctl()** call is equivalent to the TI message type **T_optmgmt_req**. The STREAMS message that the module returns in response to the successful completion of the **TI_OPTMGMT ioctl()** call is equivalent to the TI message type **T_optmgmt_ack**.

RETURN VALUES

If the **timod** module returns an error for an **ioctl()** call, the lower 8 bits of the return value will be one of the TI error codes defined in the **<tiuser.h>** header file. If the TI error is of the type TSYERR, then the second 8 bits of the return value will contain an error as defined in the **<errno.h>** header file. The STREAMS message that the module issues when an **ioctl()** call results in an error is equivalent to the TI message type **T_error_ack**.

FILES

<xti.h> defines the error codes for XTI functions.

<tiuser.h> defines the error codes for TI functions.

<tihdr.h> defines the message types for TI functions.

<errno.h> defines the error codes for system errors.

SEE ALSO

ioctl(2), t_open(3), streamio(7), tirdwr(7).

NAME

tirdwr - STREAMS module for reads and writes by Transport Interface users

DESCRIPTION

The **tirdwr** module is a STREAMS module that provides a transport user supporting the Transport Interface (TI) with an alternate interface to a transport protocol provider supporting TI. This alternate interface allows the transport user to communicate with the transport protocol provider using the **read()** and **write()** functions. It can also continue to use the **putmsg()** and **getmsg()** functions, but these functions will only transfer data messages between the user process and device stream. **getpmsg()** and **putpmsg()** should not be used with **tirdwr**.

The user places the **tirdwr** module on a device stream by calling the STREAMS **I_PUSH ioctl()** function. **tirdwr** is an alternative interface to *timod*(7). If **timod** has been pushed onto the stream, the user should use the **I_POP ioctl** to remove the **timod** module from the stream before pushing **tirdwr**. The **tirdwr** module should only be pushed onto streams which are terminated by transport providers which conform to the Transport Interface. Once the module has been pushed on the device stream the user cannot make further calls to TI functions. If the user attempts to do this, an error occurs on the stream. After the error is detected, subsequent calls fail with **errno** set to [EPROTO]. The user removes the **tirdwr** module from a device stream by calling the STREAMS **I_POP ioctl()** function.

Module Behavior When Pushed and Popped

When the **tirdwr** module is pushed on a device stream, it checks any existing messages that are destined for the user to determine their message type. If existing messages are regular data messages, it forwards the messages to the user. It ignores any messages related to process management, such as messages that generate signals to the user. If any other messages are present, it returns an error to the user request with **errno** set to [EPROTO].

When the **tirdwr** module is popped from a device stream, it checks whether an orderly release indication has been previously received from the transport protocol provider. If an orderly release indication was received, it sends an orderly release request to the remote side of the transport connection. The **tirdwr** module also acts this way when the device stream is closed.

Module Behavior for Reads and Writes

When the **tirdwr** module receives messages from the transport protocol provider that do not contain a control part (see the *putmsg*(2) and *getmsg*(2) reference pages), it transparently passes the messages to its upstream neighbor. The exception is for zero-length data messages, where the module frees the message and does not pass them to its upstream neighbor.

When the module receives messages from the transport protocol provider that contain a control part, it takes one of the following actions:

For data messages with a control part, it removes this part, then passes the message to its upstream neighbor.

For messages that represent expedited data, it generates an error. Further system calls will fail with **errno** set to [EPROTO].

For messages that represent an orderly release indication from the transport protocol provider, it generates a zero-length data message, indicating the End-of-File (EOF), and sends this message upstream to the reading process. The original message containing the orderly release indication is freed.

For messages that represent an abortive disconnect indication from the transport protocol provider, it causes all further **write()** and **putmsg()** calls to fail with **errno** set to [ENXIO]. Subsequent **read()** and **getmsg()** calls will return zero-length data messages indicating the End-of-File (EOF), once all previous data has been read.

For all other messages, it generates an error, and further calls will fail with **errno** set to [EPROTO].

SEE ALSO

getmsg(2), putmsg(2), read(2), write(2), t_open(3), streamio(7), timod(7).

NAME

tty - controlling terminal interface

DESCRIPTION

The file **/dev/tty** is, in each process, a synonym for the control terminal associated with the process group of that process, if any. It is useful for programs or shell sequences that need to be sure of writing messages on the terminal no matter how output has been redirected. It can also be used for programs that demand the name of a file for output, when typed output is desired, and it is tiresome to find out what terminal is currently in use.

FILES

/dev/tty

/dev/tty*

SEE ALSO

termio(7).

STANDARDS CONFORMANCE

tty: AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

t

NAME

UDP - Internet User Datagram Protocol

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

s = socket(AF_INET, SOCK_DGRAM, 0);
s = socket(AF_INET6, SOCK_DGRAM, 0);
```

DESCRIPTION

UDP is a simple, unreliable datagram protocol used to support the **SOCK_DGRAM** socket type for the internet protocol family. UDP sockets are connectionless, and are normally used with the **sendto()** and **recvfrom()** calls (see *send*(2) and *recv*(2). The **connect()** call can also be used to simulate a connection (see *connect*(2). When used in this manner, it fixes the destination for future transmitted packets (in which case the **send()** or **write()** system calls can be used), as well as designating the source from which packets are received. The **recv()** and **read()** calls can be used at any time if the source of the message is unimportant.

UDP address formats are identical to those used by TCP. In particular, UDP requires a port identifier in addition to the normal Internet address format. Note that the UDP port domain is separate from the TCP port domain (in other words, a UDP port cannot be connected to a TCP port).

The default send buffer size for UDP sockets is 65535 bytes. The default receive buffer size for UDP sockets is 2147483647 bytes. The send and receive buffer sizes for UDP sockets can be set by using the **SO_SNDBUF** and **SO_RCVBUF** options of the **setsockopt()** system call or the **XTI_SNDBUF** and **XTI_RCVBUF** options of the **t_optmgmt()** system call. The maximum size for these buffers is 2147483647 bytes. The maximum receive buffer size may be lowered using the **ndd** parameter **udp_recv_hiwater_max**.

The maximum message size for a UDP datagram socket is limited by the lesser of the maximum size of an IP datagram and the size of the UDP datagram socket buffer. The maximum size of an IP datagram limits the maximum message size of a UDP message to 65507 bytes. Therefore, using the maximum socket buffer size will allow multiple maximum-sized messages to be placed on the send queue. The default inbound and outbound message size limit for a UDP datagram socket is 65535 bytes.

The maximum message size for a UDP broadcast is limited by the MTU size of the underlying link.

ERRORS

One of the following errors may be returned in **errno** if a socket operation fails. For a more detailed list of errors, see the man pages for specific system calls.

[EISCONN]	Attempt to send a datagram with the destination address specified, when the socket is already connected.
[ENOBUFS]	No buffer space is available for an internal data structure.
[EADDRINUSE]	Attempt to create a socket with a port which has already been allocated.
[EADDRNOTAVAIL]	Attempt to create a socket with a network address for which no network interface exists.

u

AUTHOR

The socket interfaces to UDP were developed by the University of California, Berkeley.

SEE ALSO

ndd(1M). getsockopt(2), recv(2), send(2), socket(2), t_open(3), t_optmgmt(3) inet(7F), socket(7),

RFC 768	User Datagram Protocol
RFC 1122	Requirements for Internet hosts

NAME

UNIX - local communication domain protocol

SYNOPSIS

```
#include <sys/types.h>
#include <sys/un.h>
```

DESCRIPTION

The local communication domain protocol, commonly referred to in the industry as the **Unix domain protocol**, utilizes the path name address format and the **AF_UNIX** address family. This protocol can be used as an alternative to the Internet protocol family (TCP/IP or UDP/IP) for communication between processes executing on the same node. It has a significant throughput advantage when compared with local IP loopback, due primarily to its much lower code execution overhead. Data is looped back at the protocol layer (OSI Level 4), rather than at the driver layer (OSI Level 2).

Only **SOCK_STREAM** is supported in the **AF_UNIX** address family.

The HP-UX implementation of the local communication domain protocol does not support the **MSG_OOB** flag in **recv()** (see *recv*(2)) and **send()** (see *send*(2)).

Addressing

AF_UNIX socket addresses are path names. They are limited to 92 bytes in length, including a terminating null byte. Calls to **bind()** to an **AF_UNIX** socket utilize an addressing structure called **struct** sockaddr_un (see *bind*(2)). Pointers to this structure should be used in all **AF_UNIX** socket system calls wherever they require a pointer to a **struct sockaddr**.

The include file <**sys/un.h**> defines this addressing structure. Within this structure are two notable fields. The first is *sun_family*, which must be set to **AF_UNIX**. The next is *sun_path*, which is the null-terminated character string that specifies the path name of the file associated with the socket (for example, **/tmp/mysocket**).

Only the passive (listening) socket must bind to an address. The active socket connects to that address, but it does not need an address of its own.

For additional information on using **AF_UNIX** sockets for interprocess communication, refer to the BSD Sockets Interface Programmer's Guide.

Socket Buffer Size

For stream and datagram sockets, the maximum send and receive buffer size is 262142 bytes. The default buffer size is 32768 bytes. The send and receive buffer sizes can be altered by using the **SO_SNDBUF** and **SO_RCVBUF** options of the **setsockopt()** system call. Refer to *getsockopt*(2) for details.

AUTHOR

AF_UNIX was developed by the University of California, Berkeley.

SEE ALSO

getsockopt(2), socket(2).

NAME

VLAN - virtual local area network

DESCRIPTION

This manpage provides a brief overview of VLAN (virtual LAN) technology.

VLANs are logical, or **virtual**, network segments that can span multiple physical network segments. A primary benefit of VLANs is that they can isolate broadcast and multicast traffic by determining which destinations should receive that traffic, thereby making better use of switch and end-station resources.

Logical separation using VLAN allows for the logical grouping of PCs, servers and other network resources to behave as if they were connected to the same, physical segment, even if they are not.

HP-UX VLAN is an implementation of IEEE 802.1p/Q standards.

VLAN interfaces can be configured in HP-UX servers using the command **nwmgr** (see *nwmgr_vlan*(1M)) or **lanadmin** (see *lanadmin_vlan*(1M)). HP recommends that you use **nwmgr** for HP-UX Release 11i Version 3 and forward. Interfaces can also be configured using the web-based management tool HP-UX System Management Homepage (HP SMH).

Each VLAN interface created is assigned a VLAN PPA (VPPA) that is unique across the system and a VLAN ID, that identifies the virtual LAN it is part of. The VLAN ID is unique on the interface on which the VLAN interface is created.

WARNINGS

The **lanadmin**, **lanscan**, and **linkloop** commands are deprecated. These commands will be removed in a future HP-UX release. HP recommends the use of replacement command *nwmgr*(1M) to perform all network interface-related tasks.

SEE ALSO

lanadmin(1M), lanadmin_vlan(1M), lanscan(1M), nwmgr(1M), nwmgr_vlan(1M), smh(1M).

HP-UX VLAN Administrator's Guide

IEEE 802.1p, IEEE 802.1Q

NAME

xopen_networking - X/Open Networking Interfaces

DESCRIPTION

X/Open has defined **Sockets** and **IP Address Resolution** interfaces in *X/Open CAE Specification, Networking Services, Issue 4* (UNIX 95), *X/Open CAE Specification, Networking Services, Issue 5* (UNIX 98), and *The Single UNIX Specification, Version 3, System Interfaces* (UNIX 03).

X/Open has also defined XTI in *X/Open CAE Specification, Networking Services, Issue 4* (UNIX 95) and *X/Open CAE Specification, Networking Services, Issue 5* (UNIX 98). Beginning in UNIX 03, XTI is no longer part of *The Single UNIX Specification*.

For more information on the specifications or a detailed description of the X/Open Networking Interfaces, please refer to the above specifications at **The Open Group** website, **http://www.opengroup.org**.

Prior to HP-UX 11i v3, HP-UX is certified to UNIX 95 on PA-RISC and Integrity systems. Beginning with HP-UX 11i v3, HP-UX is certified to UNIX 95 on PA-RISC systems and to UNIX 95 and UNIX 03 on Integrity systems.

COMPILATION ENVIRONMENT

There are two ways to obtain X/Open Sockets functionality:

Method A is in compliance with X/Open compilation specification.

Method B slightly deviates from X/Open compilation specification. However, Method B allows a program to include both objects compiled to X/Open Sockets specification and objects compiled to BSD Sockets specification.

Either **cc**, **c89** or **c99** utilities can be used. Refer to *cc*(1) for details. Also note certain features in **UNIX 03** are only available if **c99** is used. For example, the "restrict" qualifier for pointers is only available if c99 is used.

Method A) Strict Compliance Method

An X/Open conforming application is one that has all its parts compiled and built according to X/Open specifications. For such conforming applications, this compilation method would be appropriate.

Compilation

UNIX 03

Applications should ensure that the feature test macro **_XOPEN_SOURCE** is defined with the value **600**. To ensure portability, applications should define the macro either on the compilation command line, or at the beginning of each source module prior to the inclusion of any headers.

For example, to compile a 64 bit object using **HP ANSI Compiler**:

```
c99 +DD64 -D_XOPEN_SOURCE=600 -c main.c -o main.o
c99 +DD64 -D_XOPEN_SOURCE=600 -c routines.c -o routines.o
```

UNIX 95

Applications should ensure that the feature test macros **_XOPEN_SOURCE** and **_XOPEN_SOURCE_EXTENDED** are defined. To ensure portability, applications should define the macros either on the compilation command line, or at the beginning of each source module prior to the inclusion of any headers.

For example, to compile a 64 bit object using **HP ANSI Compiler**:

```
c89 +DD64 -D_XOPEN_SOURCE -D_XOPEN_SOURCE_EXTENDED -c main.c -o main.o
c89 +DD64 -D_XOPEN_SOURCE -D_XOPEN_SOURCE_EXTENDED -c routines.c -o routines.o
```

Linkage

Link the program objects with **Xnet** library.

For example:

```
ld main.o routines.o -lxnet -lc -o prog
```

Note if the **C** library is also specified in the link line, the **Xnet** library has to be specified before the **C** library. Otherwise, X/Open Sockets calls would have been resolved to BSD Sockets functions in the **C** library instead of X/Open Sockets functions in the **Xnet** library.

Method B) Alternative Method

HP-UX provides two styles of Sockets API:

- default BSD Sockets
- X/Open Sockets

These two styles of Sockets API have the same function names but they have differences in semantics and argument types. For example, the *optlen* field in X/Open **getsockopt()** is **size_t** type, while BSD **getsockopt()** is **int** type. In 64 bit mode, **size_t** is 64 bit and *int* is still 32 bit.

Linking objects compiled to X/Open Sockets specification and objects compiled to BSD Sockets specification in the same program using the linkage method in method A would erroneously resolve BSD Sockets calls to X/Open Sockets functions in the **Xnet** library. As a result, the program may result in application core dumps or unexpected Socket errors when it is run. These symptoms commonly occur when BSD Sockets **accept()**, **getpeername()**, **getsockname()**, **getsockopt()**, **recvfrom()**, **sendmsg()**, and **recvmsg()** are called.

For such mixed program configuration, the compilation and linkage methods described below in *Compilation* should be used.

Compilation

Define **_HPUX_ALT_XOPEN_SOCKET_API**, in addition to either defining **_XOPEN_SOURCE=600** in **UNIX 03** or **_XOPEN_SOURCE** and **_XOPEN_SOURCE_EXTENDED** in **UNIX 95**.

For example to compile a 64-bit X/Open Sockets object and a 64-bit BSD Sockets object using **HP ANSI Compiler**:

UNIX 03

```
c99 +DD64 -D_XOPEN_SOURCE=600 -D_HPUX_ALT_XOPEN_SOCKET_API -c main.c -o main.o
```

UNIX 95

```
c89 +DD64 -D_XOPEN_SOURCE -D_XOPEN_SOURCE_EXTENDED -D_HPUX_ALT_XOPEN_SOCKET_API -c main.c -o main.o
```

BSD Sockets

```
cc -Ae +DD64 -c routines.c -o routines.o
```

With this method, X/Open Sockets calls are remapped by the static Sockets functions in <sys/socket.h> to an alternative set of X/Open Sockets functions in *C* library. This alternative set has a prefix **_xpg_** in its function names, for example, **_xpg_getsockopt()**.

Because the alternative set has different function names, X/Open Sockets calls are not confused with BSD Sockets calls at link time.

Other than the naming difference, this alternative set is identical to the X/Open Sockets functions in *Xnet* library. Other than adding an additional macro, **_HPUX_ALT_XOPEN_SOCKET_API**, this compilation method is compliant to X/Open specifications.

Linkage

Link with *C* library instead of *Xnet* library. *Xnet* library should not be included in the application link line.

X

For example:

```
ld main.o routines.o -lc -o prog
```

Because *Xnet* library is not in the link line, BSD Sockets calls are not erroneously resolved to X/Open Sockets functions in *Xnet* library.

FUTURE DIRECTION

Method B might become the default method in a future release. At that time, **_HPUX_ALT_XOPEN_SOCKET_API** would be defined by default.

AUTHOR

X/Open **XTI**, **Sockets** and **IP Address Resolution** interfaces were developed by HP and X/Open Company Limited.

SEE ALSO

XTI:

t_accept(3), t_alloc(3), t_bind(3), t_close(3), t_connect(3), t_error(3), t_free(3), t_getinfo(3), t_getprotaddr(3), t_getstate(3), t_listen(3), t_look(3), t_open(3), t_optmgmt(3), t_rcv(3), t_rcvconnect(3), t_rcvdis(3), t_rcvrel(3), t_rcvudata(3), t_rcvuderr(3), t_snd(3), t_snddis(3), t_sndrel(3), t_sndudata(3), t_strerror(3), t_sync(3), t_unbind(3).

Sockets:

accept(2), bind(2), close(2), connect(2), fcntl(2), fgetpos(3S), fsetpos(3S), ftell(3S), getpeername(2), getsockname(2), getsockopt(2), listen(2), lseek(2), poll(2), read(1), recv(2), recvfrom(2), recvmsg(2), select(2), send(2), sendmsg(2), sendto(2), setsockopt(2), shutdown(2), sockatmark(3N), socket(2), socketpair(2), write(1).

IP Address Resolution:

gethostname(2), endhostent(3N), endnetent(3N), endprotoent(3N), endservent(3N), freeaddrinfo(3N), gai_strerror(3N), getaddrinfo(3N), gethostbyaddr(3N), getnameinfo(3N), getnetbyaddr(3N), getprotobynumber(3N), getservbyport(3N), htonl(3N), if_freenameindex(3N), if_indextoname(3N), if_nameindex(3N), if_nametoindex(3N), inet_addr(3N), ntohl(3N), sethostent(3N), setnetent(3N), setprotoent(3N), setservent(3N).

X

NAME

zero - /dev/zero special file

DESCRIPTION

/dev/zero is a zero special file. Reads from a zero special file always return characters whose value is 0 (\0 characters).

Data written on a zero special file is discarded or ignored.

Seeks on a zero special file always succeed.

When **/dev/zero** is memory mapped by calling **mmap()**, the associated memory object behaves as a **MAP_ANONYMOUS** object. It is initialized to all zeros. Writes to the object modify the contents of the object which are observed by subsequent reads to this object.

Both **MAP_SHARED** and **MAP_PRIVATE** **mmap()** are allowed.

When it is mapped shared, the memory object can be shared only with the descendants of the current process. Modifications made to the **MAP_SHARED** object are visible only to the process and its descendants.

When it is mapped private, any modifications done after **fork()** are visible only to the process.

EXAMPLES

In the following example, the buffer **buf** is filled with **len** \0 characters.

```
fildes = open("/dev/zero",...)
read(fildes, buf, len)
```

In the following example, the process now has a range of **len** \0 characters at memory location **address**:

```
fildes = open("/dev/zero",...)
address = mmap(0, len, PROT_READ | PROT_WRITE,
         MAP_PRIVATE, fildes, any_offset)
```

FILES

/dev/zero

SEE ALSO

mmap(2), null(7).

Section 9
General Information

Section 9
General Information

NAME

intro - introduction to HP-UX general information section

DESCRIPTION

This section contains general information about HP-UX, including an introduction to HP-UX and the operating system and a glossary of common HP-UX terms.

SEE ALSO

glossary(9), introduction(9).

Web access to HP-UX documentation at **`http://docs.hp.com`**.

NAME

glossary - description of common HP-UX terms

DESCRIPTION

HP-UX and other UNIX-like systems use a specialized vocabulary in which certain words and terms have very specific meanings. This glossary is intended as an aid in promoting exactness in use of these specialized terms whose meanings sometimes differ from those that might be encountered in other environments. References to other HP-UX documentation are included as appropriate.

Entities in *italics* with a following parenthesized roman number (sometimes with a capital letter), such as *sh*(1), *wait*(2), or *fopen*(3S) refer to entries in the other sections of this manual. Items in **bold face** refer to other entries in this glossary. Items in `computer font` (**bold face** in the online manpages) are literals, such as file names and environment variables. Any italicized manual names refer to separate manuals that are either included with your system or available separately.

The definitions specifically reflect the HP-UX operating system, although some terms and definitions are also derived from those in the emerging IEEE POSIX standards and the *X/Open Portability Guide*. Differences in wording exist to more specifically reflect the characteristics of the HP-UX system.

GLOSSARY ENTRIES

. (dot)
A special file name that refers to the **current directory**. It can be used alone or at the beginning of a directory path name. See also **path name resolution**. The **dot** also functions as a special command in the POSIX, Bourne, and Korn shells, and has special meaning in text editors and formatters, in parsing regular expressions and in designating file names.

.. (dot-dot)
A special file name that refers to the **parent directory**. If it begins a **path name**, **dot-dot** refers to the parent of the current directory. If it occurs in a path name, **dot-dot** refers to the parent directory of the directory preceding **dot-dot** in the path name string. As a special case, **dot-dot** refers to the current directory in any directory that has no parent (most often, the **root directory**). See also **path name resolution**.

`.o` (dot-oh)
The suffix customarily given to a relocatable object file. The term **dot-oh file** is sometimes used to refer to a relocatable object file. The format of such files is sometimes called **dot-oh format**. See *a.out*(4).

`a.out`
The name customarily given to an executable object code file on HP-UX. The format is machine-dependent, and is described in *a.out*(4) for each implementation. Object code that is not yet linked has the same format, but is referred to as a `.o` (**dot-oh**) file. `a.out` is also the default output file name used by the linker, *ld*(1).

absolute path name
A path name beginning with a slash (`/`). It indicates that the file's location is given relative to the **root directory** (`/`), and that the search begins there.

access
The process of obtaining data from or placing data in storage, or the right to use system resources. Accessibility is governed by three process characteristics: the effective user ID, the effective group ID, and the group access list. The *access*(2) system call determines accessibility of a file according to the bit pattern contained in its *amode* parameter, which is constructed to read, write, execute or check the existence of a file. The *access*(2) system call uses the **real user ID** instead of the **effective user ID** and the **real group ID** instead of the **effective group ID**.

access groups
The group access list is a set of **supplementary group IDs** used in determining resource accessibility. Access checks are performed as described below in **file access permissions**.

access mode
An access mode is a form of access permitted to a file. Each implementation provides separate read, write, and execute/search access modes.

a

address

A number used in information storage or retrieval to specify and identify memory location. An **address** is used to mark, direct, indicate destination, instruct or otherwise communicate with computer elements.

In mail, **address** is a data structure whose format can be recognized by all elements involved in transmitting information. On a local system, this might be as simple as the user's **login** name, while in a networked system, **address** specifies the location of the resource to the network software.

In a text editor (such as **vi**, **ex**, **ed**, or **sed**), an **address** locates the line in a file on which a given instruction is intended.

For **adb**, the **address** specifies at what assembly-language instruction to execute a given command.

In disk utilities such as **fsdb**, **address** might refer to a raw or **block special file**, the **inode** number, **volume header**, or other file attribute.

In the context of peripheral devices, **address** refers to a set of values that specify the location of an I/O device to the computer. The exact details of the formation of an address differ between systems.

address space

The range of memory locations to which a process can refer.

affiliation

See **terminal affiliation**.

agile addressing

An addressing scheme where an address or path to a logical unit that is independent of the physical path. See *intro*(7) for more information.

appropriate privileges

Each implementation provides a means of associating privileges with a process for function calls and function call options requiring special privileges. In the HP-UX system, **appropriate privileges** refers either to superuser status or to a privilege associated with privilege groups (see *setprivgrp*(1M)).

archive

A file comprised of the contents of other files, such as a group of object files (that is, **.o**) used by the linker, *ld*(1)). An archive file is created and maintained by *ar*(1) or similar programs, such as *tar*(1) or *cpio*(1). An **archive** is often called a **library**.

ASCII

An acronym for American Standard Code for Information Interchange. ASCII is the traditional System V coded character set and defines 128 characters, including both control characters and graphic characters, each of which is represented by 7-bit binary values ranging from 0 through 127 decimal.

background process group

Any process group that is a member of a session which has established a connection with a controlling terminal that is not in the foreground process group.

backup

The process of making a copy of all or part of the file system in order to preserve it, in case a system crash occurs (usually due to a power failure, hardware error, etc.). This is a highly recommended practice.

block

(1) The fundamental unit of information HP-UX uses for access and storage allocation on a mass storage medium. The size of a block varies between implementations and between file systems. In order to present a more uniform interface to the user, most system calls and utilities use **block** to mean 512 bytes, independent of the actual block size of the medium. This is the meaning of **block** unless otherwise specified in the manual entry.

(2) On media such as 9-track tape that write variable length strings of data, the size of those strings. **Block** is often used to distinguish from **record**; a block contains several records, whereas the number of records denotes the blocking factor.

b

block special file
A special file associated with a mass storage device (such as a hard disk or tape cartridge drive) that transfers data in multiple-byte blocks, rather than by series of individual bytes (see **character special file**). **Block special files** can be mounted. A **block special file** provides access to the device where hardware characteristics of the device are not visible.

boot, boot-up
The process of loading, initializing, and running an operating system.

boot area
A portion of a mass storage medium on which the volume header and a "bootstrap" program used in booting the operating system reside. The **boot area** is reserved exclusively for use by HP-UX.

boot ROM
A program residing in ROM (Read-Only Memory) that executes each time the computer is powered up and is designed to bring the computer to a desired state by means of its own action. The first few instructions of a bootstrap program are sufficient to bring the remainder of the program into the computer from an input device and initiate functions necessary for computation. The function of the boot ROM is to run tests on the computer's hardware, find all devices accessible through the computer, and then load either a specified operating system or the first operating system found according to a specific search algorithm.

bus address
A number which makes up part of the address HP-UX uses to locate a particular device. The **bus address** is determined by a switch setting on a peripheral device which allows the computer to distinguish between two devices connected to the same interface. A **bus address** is sometimes called a "device address".

character
An element used for the organization, control, or representation of text. Characters include **graphic characters** and **control characters**.

character set
A set of characters used to communicate in a native or computer language.

character special file
A special file associated with I/O devices that transfer data byte-by-byte. Other byte-mode I/O devices include printers, nine-track magnetic tape drives, and disk drives when accessed in "raw" mode (see **raw disk**). A **character special file** has no predefined structure.

child process
A new process created by a pre-existing process via the *fork*(2) system call. The new process is thereafter known to the pre-existing process as its **child process**. The pre-existing process is the **parent process** of the new process. See **parent process** and **fork**.

clock tick
A rate used within the system for scheduling and accounting. It consists of the number of intervals per second as defined by **CLK_TCK** that is used to express the value in type **clock_t**. **CLK_TCK** was previously known as the defined constant **HZ**.

coded character set
A set of unambiguous rules that establishes a character set and the one-to-one relationship between each character of the set and its corresponding bit representation. **ASCII** is a **coded character set**.

collating element
The smallest entity used in collation to determine the logical ordering of strings (that is, the **collation sequence**). To accommodate native languages, a collating element consists of either a single character, or two or more characters collating as a single entity. The current value of the **LANG** environment variable determines the current set of collating elements.

collation
The logical ordering of strings in a predefined sequence according to rules established by precedence. These rules identify a collation sequence among the collating elements and also govern the ordering of strings consisting of multiple collating elements, to accommodate native languages.

collation sequence
The ordering sequence applied to **collating elements** when they are sorted. To accommodate native languages, **collation sequence** can be thought of as the relative order of **collating elements** as set by the current value of the **LANG** environment variable. Characters can be omitted from the collation sequence, or two or more collating elements can be given the same relative order (see *string*(3C)).

command
A directive to perform a particular task. HP-UX commands are executed through a **command interpreter** called a **shell**. HP-UX supports several shells, including the POSIX shell (*sh-posix*(1)), the C shell (*csh*(1)), and the Korn shell (*ksh*(1)). See *sh*(1) for more information about supported shells. Most commands are carried out by an executable file, called a **utility**, which might take the form of a stand-alone unit of executable object code (a program) or a file containing a list of other programs to execute in a given order (a shell script). Scripts can contain references to other scripts, as well as to object-code programs. A typical **command** consists of the utility name followed by arguments that are passed to the utility. For example, in the command, **ls mydirectory**, **ls** is the utility name and **mydirectory** is an argument passed to the **ls** utility.

command interpreter
A program which reads lines of text from standard input (typed at the keyboard or read from a file), and interprets them as requests to execute other programs. A command interpreter for HP-UX is called a **shell**. See *sh*(1) and related manual entries.

Command Set 1980
See **CS/80**.

composite graphic symbol
A graphic symbol consisting of a combination of two or more other graphic symbols in a single character position, such as a diacritical mark and a basic letter.

control character
A character other than a graphic character that affects the recording, processing, transmission, or interpretation of text. In the **ASCII** character set, **control characters** are those in the range 0 through 31, and 127. Control characters can be generated by holding down the control key (which may be labeled CTRL, CONTROL, or CNTL depending on your terminal), and pressing a character key (as you would use SHIFT). These two-key sequences are often written as, for example, Control-**D**, Ctrl-**D**, or **^D**, where **^** stands for the control key.

controlling process
The session leader that establishes the connection to the **controlling terminal**. Should the terminal subsequently cease to be a controlling terminal for this session, the session leader ceases to be the controlling process.

controlling terminal
A terminal that is associated with a session. Each session can have at most one controlling terminal associated with it and a controlling terminal is associated with exactly one session. Certain input sequences from the controlling terminal cause signals to be sent to all processes in the foreground process group associated with the controlling terminal.

Coordinated Universal Time (UTC)
See **Epoch**.

CS/80, CS-80
A family of mass storage devices that communicate with the controlling computer by means of a series of commands and data transfer protocol referred to as the **CS/80** (Command Set 1980) command set. This command set was implemented in order to provide better forward/backward compatibility between models and generations of mass storage devices as technological advances develop. Some mass storage devices support only a subset of the full **CS/80** command set, and are usually referred to as **SS/80** (Subset 1980) devices.

crash
The unexpected shutdown of a program or system. If the operating system crashes, this is a "system crash", and requires the system to be rebooted.

current directory
See **working directory**.

current working directory
See **working directory**.

daemon
A process which runs in the background, and which is usually immune to termination instructions from a terminal. Its purpose is to perform various scheduling, clean-up, and maintenance jobs. *lpsched*(1M) is an example of a **daemon**. It exists to perform these functions for line printer jobs queued by *lp*(1). An example of a permanent **daemon** (that is, one that should never die) is *cron*(1M).

data encryption
A method for encoding information in order to protect sensitive or proprietary data. For example, HP-UX automatically encrypts all users' passwords. The encryption method used by HP-UX converts ASCII text into a base-64 representation using the alphabet **.**, **/**, **0**-**9**, **A**-**Z**, **a**-**z**. See *passwd*(4) for the numerical equivalents associated with this alphabet.

default search path
The sequence of directory prefixes that *sh*(1), *time*(1), and other HP-UX commands apply in searching for a file known by an relative path name (that is, a path name not beginning with a **slash** (**/**)). It is defined by the environment variable **PATH** (see *environ*(5)). *login*(1) sets **PATH** equal to **:/usr/bin**, which means that your working directory is the first directory searched, followed by **/usr/bin**. The search path can be redefined by modifying the value of **PATH**. This is usually done in **/etc/profile**, and/or in the **.profile** file found in the home directory.

defunct process
See **zombie process**.

delta
A term used in the **Source Code Control System** (SCCS) to describe a unit of one or more textual changes to an **SCCS file**. Each time an SCCS file is edited, changes made to the file are stored separately as a **delta**. The *get*(1) command is then used to specify which deltas are to be applied to or excluded from the SCCS file, thus yielding a particular version of the file. Contrast this with the **vi** or **ed** editor, which incorporates changes into the file immediately, eliminating any possibility of obtaining a previous version of that file. A similar capability is provided by RCS files (see *rcsintro*(5)).

demon
Improper spelling of the UNIX word **daemon**.

device
A computer peripheral or an object that appears to an application as such.

device address
See **bus address**.

device file
See **special file**.

directory
A file that provides the mapping between the names of files and their contents, and is manipulated by the operating system alone. For every file name contained in a directory, that directory contains a pointer to the file's **inode**; The pointer is called a **link**. A file can have several links appearing anywhere on the same file system. Each user is free to create as many directories as needed (using *mkdir*(1)), provided that the **parent directory** of the new directory gives the permission to do so. Once a directory has been created, it is ready to contain ordinary files and other directories. An HP-UX directory is named and behaves exactly like an ordinary file, with one exception: no user (including the superuser) is allowed to write data on the directory itself; this privilege is reserved for the HP-UX operating system.

By convention, a directory contains at least two links, **.** and **..**, referred to as **dot** and **dot-dot** respectively. **.** refers to the directory itself and **..** refers to its **parent directory**. A directory containing only **.** and **..** is considered empty.

dot
See **.** (**dot**).

dot-dot
See **..** (**dot-dot**).

dot-oh
See **.o** (**dot-oh**).

dot-oh file
See **.o** (**dot-oh**).

dot-oh format
See **.o** (**dot-oh**).

downshifting
The conversion of an uppercase character to its lowercase representation.

dynamic loader
A routine invoked at process startup time that loads shared libraries into a process's address space. The dynamic loader also resolves symbolic references between a program and the shared libraries, and initializes the shared libraries' linkage tables. See *dld.sl*(5) (PA-RISC systems) or *dld.so*(5) (Itanium®-based systems) for details.

effective group ID
Every process has an **effective group ID** that is used to determine **file access permissions**. A process's **effective group ID** is determined by the file (command) that process is executing. If that file's set-group-ID bit is set (located in the mode of the file, see **mode**), the process's **effective group ID** is set equal to the file's group ID. This makes the process appear to belong to the file's group, perhaps enabling the process to access files that must be accessed in order for the program to execute successfully. If the file's set-group-ID bit is not set, the process's **effective group ID** is inherited from the process's parent. The setting of the process's **effective group ID** lasts only as long as the program is being executed, after which the process's effective group ID is set equal to its real group ID. See **group**, **real group ID**, and **set-group-ID bit**.

effective user ID
A process has an **effective user ID** that is used to determine **file access permissions** (and other permissions with respect to system calls, if the effective user ID is 0, which means superuser). A process's effective user ID is determined by the file (command) that process is executing. If that file's set-user-ID bit is set (located in the mode of the file, see **mode**), the process's effective user ID is set equal to the file's user ID. This makes the process appear to be the file's owner, enabling the process to access files which must be accessed in order for the program to execute successfully. (Many HP-UX commands which are owned by **root**, such as **mkdir** and **mail**, have their set-user-ID bit set so other users can execute these commands.) If the file's set-user-ID bit is not set, the process's effective user ID is inherited from that process's parent. See **real user ID** and **set-user-ID bit**.

end-of-file (EOF)
(1) The data returned when attempting to read past the logical end of a file via *stdio*(3S) routines. In this case, end-of-file is not properly a character.

(2) The ASCII character Ctrl-**D**.

(3) A character defined by *stty*(1) or *ioctl*(2) (see *termio*(7)) to act as end-of-file on your terminal. Usually this is Ctrl-**D**.

(4) The return value from *read*(2) that indicates end of data.

environment
The set of defined shell variables (such as **EXINIT**, **HOME**, **PATH**, **SHELL**, **TERM**, and others) that define the conditions under which user commands run. These conditions can include user terminal characteristics, home directory, and default search path. Each shell variable setting in the current process is passed on to all **child processes** that are created, provided that each shell variable setting has been exported via the **export** command (see *sh*(1)). Unexported shell variable settings are meaningful only to the current process, and any child processes created get the default settings of certain shell variables by executing **/etc/profile**, **$HOME/.profile**, or **$HOME/.login**.

d

EOF

See **end-of-file**.

Epoch

The time period beginning at 0 hours, 0 minutes, 0 seconds, **Coordinated Universal Time** (**UTC**) on January 1, 1970. Increments quantify the amount of time elapsed from the Epoch to the referenced time.

Leap seconds, which occur at irregular intervals, are not reflected in the count of seconds between the Epoch and the referenced time. (Fourteen leap seconds occurred in the years 1970 through 1988.)

FIFO special file

A type of **file**. Data written to a **FIFO** is read on a first-in-first-out basis. Other characteristics are described in *open*(2), *read*(2), *write*(2) and *lseek*(2).

file

A stream of bytes that can be written to and/or read from. A **file** has certain attributes, including permissions and type. File types include **regular file**, **character special file**, **block special file**, **FIFO special file**, network special file, **directory**, and **symbolic link**. Every file must have a **file name** that enables the user (and many of the HP-UX commands) to refer to the contents of the file. The system imposes no particular structure on the contents of a file, although some programs do. Files can be accessed serially or randomly (indexed by byte offset). The interpretation of file contents and structure is up to the programs that access the file.

file access mode

A characteristic of an **open file description** that determines whether the described file is open for reading, writing, or both. (See *open*(2).)

file access permissions

Every file in the **file hierarchy** has a set of access permissions. These permissions are used in determining whether a process can perform a requested operation on the file (such as opening a file for writing). Access permissions are established when a file is created via the *open*(2) or *creat*(2) system calls, and can be changed subsequently through the *chmod*(2) call. These permissions are read by *stat*(2) or *fstat*(2).

File access controls whether a file can be read, written, or executed. Directory files use the execute permission to control whether or not the directory can be searched.

File access permissions are interpreted by the system as they apply to three different classes of users: the **owner** of the file, the users in the file's **group**, and anyone else ("other"). Every file has an independent set of access permissions for each of these classes. When an access check is made, the system decides if permission should be granted by checking the access information applicable to the caller.

Read, write, and execute/search permissions on a file are granted to a process if any of the following conditions are met:

- The process's **effective user ID** is superuser.
- The process's **effective user ID** matches the user ID of the owner of the file and the appropriate access bit of the **owner** portion (0700) of the file mode is set.
- The process's **effective user ID** does not match the user ID of the owner of the file, and either the process's **effective group ID** matches the group ID of the file, or the group ID of the file is in the process's group access list, and the appropriate access bit of the **group** portion (070) of the file mode is set.
- The process's **effective user ID** does not match the user ID of the owner of the file, and the process's **effective group ID** does not match the group ID of the file, and the group ID of the file is not in the process's group access list, and the appropriate access bit of the "other" portion (07) of the file mode is set.

Otherwise, the corresponding permissions are denied.

file descriptor

A small unique, per-process, nonnegative integer identifier that is used to refer to a file opened for reading and/or writing. Each **file descriptor** refers to exactly one **open file description**.

A **file descriptor** is obtained through system calls such as *creat*(2), *fcntl*(2), *open*(2), *pipe*(2), or *dup*(2). The **file descriptor** is used as an argument by calls such as *read*(2), *write*(2), *ioctl*(2), and *close*(2).

The value of a **file descriptor** has a range from 0 to one less than the system-defined maximum. The system-defined maximum is the value **NOFILE** in **<sys/param.h>**.

file group class
A process is in the **file group class** of a file if the process is not the **file owner class** and if the **effective group ID** or one of the **supplementary group ID**s of the process matches the group ID associated with the file.

file hierarchy
The collection of one or more **file systems** available on a system. All **files** in these **file systems** are organized in a single hierarchical structure in which all of the nonterminal nodes are **directories**. Because multiple **links** can refer to the same **file**, the directory is properly described as a directed graph.

file name
A string of up to 14 bytes (or 255 bytes on file systems that support long file names) used to refer to an ordinary file, special file, or directory. The byte values NUL (null) and slash (**/**) cannot be used as characters in a file name. Note that it is generally unwise to use *****, **?**, **,**, **[**, or **]** as part of file names because the shell attaches special meaning to these characters (see *sh*(1), *csh*(1), or *ksh*(1)). Avoid beginning a file name with **-**, **+**, or **=**, because to some programs, these characters signify that a command argument follows. A file name is sometimes called a path name component. Although permitted, it is inadvisable to use characters that do not have a printable graphic on the hardware you commonly use, or that are likely to confuse your terminal.

file name portability
File names should be constructed from the **portable file name character set** because the use of other characters can be confusing or ambiguous in certain contexts.

file offset
The file offset specifies the position in the file where the next I/O operation begins. Each **open file description** associated with either a regular file or special file has a **file offset**. There is no file offset specified for a **pipe** or **FIFO**.

file other class
A process is in the **file other class** if the process is not in the **file owner class** or **file group class**.

file owner class
A process is in the **file owner class** if the **effective user ID** of the process matches the user ID of the file.

file permission bits
See **permission bits**.

file pointer
A data element obtained through any of the *fopen*(3S) standard I/O library routines that "points to" (refers to) a file opened for reading and/or writing, and which keeps track of where the next I/O operation will take place in the file (in the form of a byte offset relative to the beginning of the file). After obtaining the file pointer, it must thereafter be used to refer to the open file when using any of the standard I/O library routines. (See *stdio*(3S) for a list of these routines.)

file serial number
A file-system-unique identifier for a given file, also known as the file's **inode number**. Each **file serial number** identifies exactly one **inode**. **File serial numbers** are not necessarily unique across **file systems** in the **file hierarchy**.

file status flags
Part of an **open file description**. These flags can be used to modify the behavior of system calls that access the file described by the **open file description**.

file system
A collection of **files** and supporting data structures residing on a mass storage volume. A file system provides a name space for **file serial numbers** referring to those files. Refer to the System Administrator manuals supplied with your system for details concerning file system implementation and maintenance.

file times update

Each file has three associated time values that are updated when file data is accessed or modified, or when the file status is changed. These values are returned in the file characteristics structure, as described in **<sys/stat.h>**. For each function in HP-UX that reads or writes file data or changes the file status, the appropriate time-related files are noted as "marked-for-update". When an update point occurs, any marked fields are set to the current time and the update marks are cleared. One such update point occurs when the file is no longer open for any process. Updates are not performed for files on **read-only file systems**.

filter

A command that reads data from the standard input, performs a transformation on the data, and writes it to the standard output.

foreground process group

Each session that has established a connection with a controlling terminal has exactly one process group of the session as a foreground process group of that controlling terminal. The foreground process group has certain privileges when accessing its controlling terminal that are denied to background process groups. See *read*(2) and *write*(2).

f

foreground process group ID

The process group ID of the foreground process group.

fork

An HP-UX system call (see *fork*(2)), which, when invoked by an existing process, causes a new process to be created. The new process is called the **child process**; the existing process is called the **parent process**. The child process is created by making an exact copy of the parent process. The parent and child processes are able to identify themselves by the value returned by their corresponding **fork** call (see *fork*(2) for details).

graphic character

A character other than a control character that has a visual representation when hand-written, printed, or displayed.

group

See **group ID**.

group ID

Associates zero or more users who must all be permitted to access the same set of files. The members of a group are defined in the files **/etc/passwd** and **/etc/logingroup** (if it exists) via a numerical group ID that must be between zero and **UID_MAX**, inclusive. Users with identical group IDs are members of the same group. An ASCII group name is associated with each group ID in the file **/etc/group**. A group ID is also associated with every file in the **file hierarchy**, and the mode of each file contains a set of permission bits that apply only to this group. Thus, if you belong to a group that is associated with a file, and if the appropriate permissions are granted to your group in the file's mode, you can access the file. When the identity of a group is associated with a process, a group ID value is referred to as a **real group ID**, an **effective group ID**, a **supplementary group ID**, or a **saved group ID**. See also **privileged group** and **set-group-ID bit**.

group access list

A set of **supplementary group ID**s used in determining resource accessibility. Access checks are performed as described in **file access permissions**.

hardware path

A numeric string associated to a system component (bus, card, attached I/O device, and so on) and providing information related to the component location.

hierarchical directory

A directory (or file system) structure in which each directory can contain other directories as well as files.

home directory

The directory name given by the value of the environment variable **HOME**. When you first log in, *login*(1) automatically sets **HOME** to your **login directory**. You can change its value at any time. This is usually done in the **.profile** file contained in your **login directory**. Setting **HOME** does not affect your **login**

directory; it simply gives you a convenient way of referring to what is probably your most commonly used directory.

host name
A string of bytes that uniquely identifies the system in the network. The host name for your system can be viewed and/or set with the *hostname*(1) command. More information can be found in the *hostname*(5) manpage. See also **node name**.

image
The current state of your computer (or your portion of the computer, on a multiuser system) during the execution of a command. Often thought of as a "snapshot" of the state of the machine at any particular moment during execution.

init
A **system process** that performs initialization, is the ancestor of every other process in the system, and is used to start **login** processes. **init** usually has a **process ID** of **1**. See *init*(1M).

interleave factor
A number that determines the order in which sectors on a mass storage medium are accessed. It can be optimized to make data acquisition more efficient.

inode
An **inode** is a structure that describes a file and is identified in the system by a **file serial number**. Every file or directory has associated with it an **inode**. Permissions that specify who can access the file and how are kept in a 9-bit field that is part of the **inode**. The **inode** also contains the file size, the user and group ID of the file, the number of links, and pointers to the disk blocks where the file's contents can be found. Each connection between an **inode** and its entry in one or more directories is called a **link**.

inode number
See **file serial number**.

Internal Terminal Emulator (ITE)
The "device driver" code contained in the HP-UX kernel that is associated with the computer's built-in keyboard and display or with a particular keyboard and display connected to the computer, depending on the Series and Model of system processor. See **system console** and the System Administrator manuals supplied with your system for details.

internationalization
The concept of providing software with the ability to support the **native language**, **local customs**, and **coded character set** of the user.

interrupt signal
The signal sent by **SIGINT** (see *signal*(2)). This signal generally terminates whatever program you are running. The key which sends this signal can be redefined with *ioctl*(2) or *stty*(1) (see *termio*(7)). It is often the ASCII DEL (rubout) character (the DEL key) or the BREAK key. Ctrl-**C** is often used instead.

intrinsic
See **system call**.

I/O redirection
A mechanism provided by the HP-UX shell for changing the source of data for standard input and/or the destination of data for standard output and standard error. See *sh*(1).

ITE
See **Internal Terminal Emulator**.

job control
Job control allows users to selectively stop (suspend) execution of processes and continue (resume) their execution at a later time.

The user employs this facility via the interactive interface jointly supplied by the system terminal driver and certain shells (see *sh*(1)). The terminal driver recognizes a user-defined "suspend character", which causes the current foreground process group to stop and the user's job control shell to resume. The job control shell provides commands that continue stopped process groups in either the foreground or background. The terminal driver also stops a background process group when any member of the background process group attempts to read from or write to the user's terminal. This allows the user to finish or suspend the **foreground process group** without interruption and continue the stopped **background process group** at a more convenient time.

See *stty*(1), *sh*(1), and related shell entries for usage and installation details, and the shell entries plus *signal*(2) and *termio*(7) for implementation details.

kernel
The HP-UX operating system. The kernel is the executable code responsible for managing the computer's resources, such as allocating memory, creating processes, and scheduling programs for execution. The kernel resides in RAM (random access memory) whenever HP-UX is running.

LANG
An environment variable used to inform a computer process of the user's requirements for **native language**, **local customs**, and **coded character set**.

legacy device special file
A special file associated with an I/O device (tape, disk, and so on), locked to a particular physical **hardware path**, containing hardware path information such as SCSI bus, target, and LUN in the device file name and minor number. See *intro*(7) for more information.

j

legacy hardware path
A hardware path following the legacy format conventions, that is, a series of bus-nexus addresses separated by **/** (slash) characters, leading to a host bus adapter (HBA). Beneath the HBA, additional address elements are separated by **.** (period) characters. All elements are represented in decimal. See *intro*(7) for more information.

library
A file containing a set of subroutines and variables that can be accessed by user programs. Libraries can be either archives or shared libraries. For example, **/usr/lib/libc.a** and **/usr/lib/libc.sl** are libraries containings all functions of Section 2 and all functions of Section 3 that are marked (3C) and (3S) in the *HP-UX Reference*. Similarly, **/usr/lib/libm.a** and **/usr/lib/libm.sl** are libraries containing all functions in Section 3 that are marked (3M) in the *HP-UX Reference*. See *intro*(2) and *intro*(3C).

LIF
See **Logical Interchange Format**.

line
A sequence of text characters consisting of zero or more nonnewline characters plus a terminating newline character.

link
Link is a synonym for **directory entry**. It is an object that associates a file name with any type of file. The information constituting a **link** includes the name of the file and where the contents of that file can be found on a mass storage medium. One physical file can have several links to it. Several directory entries can associate names with a given file. If the links appear in different directories, the file may or may not have the same name in each. However, if the links appear in one directory, each link must have a unique name in that directory. Multiple links to directories are not allowed (except as created by a user with appropriate privileges). See *ln*(1), *link*(2), *unlink*(2), and **symbolic link**.

Also, to prepare a program for execution; see **linker**.

link count
The number of directory entries that refer to a particular file.

linker
A program that combines one or more object programs into one program, searches libraries to resolve user program references, and builds an executable file in **a.out** format. This executable file is ready to be executed through the program loader, *exec*(2). The linker is invoked with the *ld*(1) command. The linker is often called a **link editor**.

local customs
The conventions of a geographical area or territory for such things as date, time and currency formats.

localization
The process of adapting existing software to meet the local language, customs, and character set requirements of a particular geographical area.

Logical Interchange Format (LIF)
A standard format for mass storage implemented on many Hewlett-Packard computers to aid in media transportability. See *lif*(4) for more detail.

login
The process of gaining access to HP-UX. This consists of successful execution of the login sequence defined by *login*(1), which varies depending on the system configuration. It requests a **login** name and possibly one or more passwords.

login directory
The directory in which you are placed immediately after you log in. This directory is defined for each user in the file **/etc/passwd**. The shell variable **HOME** is set automatically to your **login directory** by *login*(1) immediately after you log in. See **home directory**.

LUN
LUN refers to an end device, such as a disk or tape or a piece of logical storage in a disk array (mass storage term). Also known as a Logical Unit (LU).

LUN hardware path
A virtualized path that can represent multiple paths to a single mass storage device. It starts with a virtual bus-nexus (known as the **virtual root node**) with an address of 64000. Addressing beneath that virtual root node consists of a virtual bus address and a virtual LUN identifier, delimited by **/** (slash) characters. See *intro*(7) for more information.

lunpath hardware path
A hardware path to a LUN. It is composed of a series of bus-nexus addresses separated by **/** (slash) characters, leading to a host bus adopter (HBA). Beneath the HBA, additional address elements are represented in hexadecimal. The first elements represent a transport-dependent target address. The final element is a LUN address, which is the 64-bit representation of the LUN identifier reported by the target. See *intro*(7) for more information.

magic number
The first word of an **a.out** format or archive file. This word contains the system ID, which states what machine (hardware) the file will run on, and the file type (executable, sharable executable, archive, etc.).

major number
A number used exclusively to create special files that enable I/O to or from specific devices. This number indicates which device driver to use for the device. Refer to *mknod*(2) and the System Administrator manual supplied with your system for details.

message catalog
Program strings, such as program messages and prompts, are stored in a **message catalog** corresponding to a particular geographical area. Retrieval of a string from a **message catalog** is based on the value of the user's **LANG** environment variable (see **LANG**).

message queue identifier (msqid)
A unique positive integer created by a *msgget*(2) system call. Each **msqid** has a message queue and a data structure associated with it. The data structure is referred to as **msqid_ds** and contains the following members:

```
struct
ipc_perm  msg_perm;   /* operation permission */
msgqnum_t msg_qnum;   /* number of msgs on q */
msglen_t  msg_qbytes; /* max number of bytes on q */
msglen_t  msg_cbytes; /* current number of bytes on q */
pid_t     msg_lspid;  /* pid of last msgsnd operation */
```

```
pid_t     msg_lrpid;  /* pid of last msgrcv operation */
time_t    msg_stime;  /* last msgsnd time */
time_t    msg_rtime;  /* last msgrcv time */
time_t    msg_ctime;  /* last change time */
                      /* Times measured in secs since */
                      /* 00:00:00 GMT, Jan. 1, 1970 */
```

Message queue identifiers can be created using *ftok*(3C).

msg_perm is a **ipc_perm** structure that specifies the message operation permission (see below). This structure includes the following members:

```
uid_t   cuid;     /* creator user id */
gid_t   cgid;     /* creator group id */
uid_t   uid;      /* user id */
gid_t   gid;      /* group id */
mode_t   mode;    /* r/w permission */
```

msg_qnum is the number of messages currently on the queue. **msg_qbytes** is the maximum number of bytes allowed on the queue. **msg_lspid** is the process id of the last process that performed a **msgsnd** operation. **msg_lrpid** is the process id of the last process that performed a **msgrcv** operation. **msg_stime** is the time of the last **msgsnd** operation, **msg_rtime** is the time of the last **msgrcv** operation, and **msg_ctime** is the time of the last *msgctl*(2) operation that changed a member of the above structure.

message operation permissions
In the *msgop*(2) and *msgctl*(2) system call descriptions, the permission required for an operation is indicated for each operation. Whether a particular process has these permissions for an object is determined by the object's permission mode bits as follows:

00400	Read by user
00200	Write by user
00060	Read, Write by group
00006	Read, Write by others

m

Read and Write permissions on a **msqid** are granted to a process if one or more of the following are true:

- The process's effective user ID is superuser.
- The process's effective user ID matches **msg_perm.**[**c**]**uid** in the data structure associated with **msqid** and the appropriate bit of the "user" portion (0600) of **msg_perm.mode** is set.
- The process's effective user ID does not match **msg_perm.**[**c**]**uid** and either the process's effective group ID matches **msg_perm.**[**c**]**gid** or one of **msg_perm.**[**c**]**gid** is in the process's group access list and the appropriate bit of the "group" portion (00060) of **msg_perm.mode** is set.
- The process's effective user ID does not match **msg_perm.**[**c**]**uid** and the process's effective group ID does not match **msg_perm.**[**c**]**gid** and neither of **msg_perm.**[**c**]**gid** is in the process's group access list and the appropriate bit of the "other" portion (06) of **msg_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

metacharacter
A character that has special meaning to the HP-UX shell, as well as to commands such as **ed**, **find**, and **grep** (see *ed*(1), *find*(1), and *grep*(1)). The set of metacharacters includes: **!**, **"**, **&**, **'**, *****, **;**, **<**, **>**, **?**, **[**, **]**, **`**, and **|**. Refer to *sh*(1) and the related shell manual entries for the meaning associated with each. See also **regular expression**.

minor number
A number that is an attribute of special files, specified during their creation and used whenever they are accessed, to enable I/O to or from specific devices. This number is passed to the device driver and is used to select which device in a family of devices is to be used, and possibly some operational modes. The exact format and meaning of the **minor number** depends both on the driver and on the addressing format (legacy or agile) being used. In legacy format, the minor number encodes path information, but in agile format, the minor number is opaque and based on the WWID.

mode
A 16-bit word associated with every file in the file system, stored in the **inode**. The least-significant 12 bits of the **mode** determine the read, write, and execute permissions for the file owner, file group, and all others, and contain the set-user-ID, set-group-ID, and sticky bits. The least-significant 12 bits can be set by the *chmod*(1) command if you are the file's owner or the superuser. These 12 bits are sometimes referred to as **permission bits**. The most-significant 4 bits specify the file type for the associated file and are set as the result of *open*(2) or *mknod*(2) system calls.

mountable file system
A removable blocked file system contained on some mass storage medium with its own root directory and an independent hierarchy of directories and files. See **block special file** and *mount*(1M).

msqid
See **message queue identifier**.

Multiplexer (MUX)
Multiplexer (MUX) is a high-speed serial communication multiple port product. It combines various signals for transmission over a single channel and provides intelligent communication functions to off-load CPU serial communication processing tasks.

multiuser state
The condition of the HP-UX operating system in which terminals (in addition to the system console) allow communication between the system and its users. By convention, multiuser run level is set at state 2, which is usually defined to contain all the terminal processes and **daemons** needed in a multiuser environment. Run levels are table driven, and are specified by *init*(1M), which sets the run level by looking at the file **`/etc/inittab`**. Do not confuse the multiuser system with the multiuser state. A multiuser system is a system which can have more than one user actively communicating with the system when it is in the multiuser state. The multiuser state removes the single-user restriction imposed by the single-user state (see **single-user state**, *inittab*(4)).

native language
A computer user's spoken or written language, such as Chinese, Dutch, English, French, German, Greek, Italian, Katakana, Korean, Spanish, Swedish, Turkish, and so on.

Network File System (NFS)
The Network File System (NFS) allows a client node to perform transparent file access over the network.

By using NFS, a client node operates on files residing on a variety of servers and server architectures, and across a variety of operating systems. File access calls on the client (such as read requests) are converted to NFS protocol requests and sent to the server system over the network. The server receives the request, performs the actual file system operation, and sends a response back to the client.

NFS operates in a stateless manner using remote procedure calls (RPC) built on top of an external data representation (XDR) protocol. The RPC protocol enables version and authentication parameters to be exchanged for security over the network.

A server grants access to a specific file system to clients by adding an entry for that file system to the server's **`/etc/dfs/dfstab`** file.

Native Language Support (NLS)
A feature of HP-UX that provides the user with internationalized software and the application programmer with tools to develop this software.

newline character
The character with an ASCII value of 10 (line feed) used to separate lines of characters. It is represented by **`\n`** in the C language and in various utilities. The terminal driver normally interprets a carriage-return/line-feed sequence sent by a terminal as a single newline character (but see *tty*(7) for full details)

NLS
See **Native Language Support**.

NLSPATH
An environment variable used to indicate the search path for message catalogs (see **message catalog**).

node name
A string of bytes which uniquely identifies the system in the local network. Unlike the **host name**, the node name cannot include domain names. It can be viewed and/or set with the *uname*(1) command. The node and host names are usually set to the same value as application programs sometimes use the node and host names interchangeably.

nonspacing characters
Characters, such as a diacritical mark or accents, that are used in combination with other characters to form composite graphic symbols commonly found in non-English languages.

open file
A file that is currently associated with a file descriptor.

open file description
A record of how a process or a group of processes is accessing a file. Each **file descriptor** refers to exactly one **open file description**, but an **open file description** can be referred to by more than one file descriptor. The **file offset**, **file status flags**, and **file access mode**s are attributes of an **open file description**.

ordinary file
A type of HP-UX file containing ASCII text (for example, program source), binary data (for example, executable code), etc. Ordinary files can be created by the user through I/O redirection, editors, or HP-UX commands.

orphan process
A **child process** that is left behind when a **parent process** terminates for any reason. The **init** process (see *init*(1M)) inherits (that is, becomes the effective parent of) all orphan processes.

orphaned process group
A process group in which the parent of every member is either itself a member of the group or is not a member of the group's session.

n

owner
The owner of a file is usually the creator of that file. However, the ownership of a file can be changed by the superuser or the current owner with the *chown*(1) command or the *chown*(2) system call. The file owner is able to do whatever he wants with his files, including remove them, copy them, move them, change their contents, etc. The owner can also change the files' modes.

parent directory
The directory one level above a directory in the **file hierarchy**. All directories except the **root directory** (**/**) have one (and only one) parent directory. The **root directory** has no parent. See also **dot** and **dot-dot**.

parent process
Whenever a new process is created by a currently-existing process (via *fork*(2)), the currently existing process is said to be the parent process of the newly created process. Every process has exactly one parent process (except the **init** process, see **init**), but each process can create several new processes with the *fork*(2) system call. The parent process ID of any process is the **process ID** of its creator.

parent process ID
A new process is created by a currently active process. The **parent process ID** of a process is the process ID of its creator for the lifetime of the creator. After the creator's lifetime has ended, the **parent process ID** is the process ID of **init**.

password
A string of ASCII characters used to verify the identity of a user. Passwords can be associated with users and groups. If a user has a password, it is automatically encrypted and entered in the second field of that user's line in the **/etc/passwd** file. A user can create or change his or her own password by using the *passwd*(1) command.

path name
A sequence of directory names separated by slashes, and ending with any file name. All file names except the last in the sequence *must* be directories. If a path name begins with a **slash** (**/**), it is an **absolute**

path name; otherwise, it is a **relative path name**. A path name defines the path to be followed through the hierarchical file system in order to find a particular file.

More precisely, a path name is a null-terminated character string constructed as follows:

<path-name>::=<file-name>|<path-prefix><file-name>|**/**
<path-prefix>::=<rtprefix>|**/**<rtprefix>
<rtprefix>::=<dirname>**/**|<rtprefix><dirname>**/**

where <file-name> is a string of one or more characters other than the ASCII slash and null, and <dirname> is a string of one or more characters (other than the ASCII slash and null) that names a directory. File and directory names can consist of up to 14 characters on systems supporting short file names and up to 255 characters on systems supporting long file names.

A **slash** (**/**) by itself names the **root directory**. Two or more slashes in succession (**////**...) are treated as a single slash.

Unless specifically stated otherwise, the null or zero-length path name is treated as though it named a nonexistent file.

path name resolution
The process that resolves a path name to a particular file in a **file hierarchy**. Multiple path names can resolve to the same file, depending on whether resolution is sought in absolute or relative terms (see below). Each file name in the path name is located in the directory specified by its predecessor (for example, in the path name fragment **a/b**, file **b** is located in directory **a**). **Path name resolution** fails if this cannot be accomplished.

If the path name begins with a slash, the predecessor of the first file name in the path name is understood to be the **root directory** of the process, and the path name is referred to as an **absolute path name**. If the path name does not begin with a slash, the predecessor of the first file name of the path name is understood to be the current working directory of the process, and the path name is referred to as a **relative path name**. A path name consisting of a single slash resolves to the root directory of the process.

path prefix
A **path name** with an optional ending **slash** that refers to a **directory**.

permission bits
The nine least-significant bits of a file's **mode** are referred to as file **permission bits**. These bits determine read, write, and execute permissions for the file's **owner**, the file's **group**, and all others. The bits are divided into three parts: owner, group and other. Each part is used with the corresponding file class of processes. The bits are contained in the file mode, as described in *stat*(5). The detailed usage of the file permission bits in access decisions is described in **file access permissions**.

persistent device special file
A device file for mass storage devices, which is associated with a LUN hardware path, and thus transparently supports **agile addressing** and multipathing. In other words, a persistent device special file is unchanged if the LUN is moved from one host bus adapter (HBA) to another, moved from one switch/hub port to another, presented via a different target port to the host, or configured with multiple hardware paths. See *intro*(7) for more information on device special files.

PIC
See **position-independent code**.

pipe
An interprocess I/O channel used to pass data between two processes. It is commonly used by the **shell** to transfer data from the standard output of one process to the standard input of another. On a command line, a pipe is signaled by a vertical bar (**|**). Output from the command to the left of the vertical bar is channeled directly into the standard input of the command on the right.

portable file name character set
The following set of graphical characters are portable across conforming implementations of IEEE Standard P1003.1:

ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
01234567890._-

The last three characters are the dot, underscore and hyphen characters, respectively. The hyphen should not be used as the first character of a portable file name.

position-independent code (PIC)
Object code that can run unmodified at any virtual address. Position-independent code can use PC-relative addressing modes and/or linkage tables. It is most often used in shared libraries, in which case the linkage tables are initialized by the dynamic loader. Position-independent code is generated when the **+z** or **+Z** compiler option is specified.

privileged groups
A **privileged group** is a group that has had a **setprivgrp** (see *getprivgrp*(2)) operation performed on it, giving it access to some system calls otherwise reserved for the superuser. See **appropriate privileges**.

process
An invocation of a program, or the execution of an image (see **image**). Although all commands and utilities are executed within processes, not all commands or utilities have a one-to-one correspondence with processes. Some commands (such as **cd**) execute within a process, but do not create any new processes. Others (such as in the case of **ls | wc -l**) create multiple processes. Several processes can be running the same program, but each can be different data and be in different stages of execution. A process can also be thought of as an **address space** and single thread of control that executes within that address space and its required system resources. A **process** is created by another process issuing the *fork*(2) function. The process that issues *fork*(2) is known as the **parent process** and the new process created by the *fork*(2) as the **child process**.

process 1
See **init**.

process group
Each process in the system is a member of a **process group**. This grouping permits the signaling of related processes. A newly created process joins the process group of its creator.

process group ID
Each process group in the system is uniquely identified during its lifetime by a **process group ID**, a positive integer less than or equal to **PIC_MAX**. A **process group ID** cannot be reused by the system until the process group lifetime ends.

p

process group leader
A **process group leader** is a process whose process ID is the same as its process group ID.

process group lifetime
A period of time that begins when a **process group** is created and ends when the last remaining process in the group leaves the group, either due to process termination or by calling the *setsid*(2) or *setpgid*(2) functions.

process ID
Each active process in the system is uniquely identified during its lifetime by a positive integer less than or equal to **PID_MAX** called a **process ID**. A process ID cannot be reused by the system until after the process lifetime ends. In addition, if there exists a process group whose process group ID is equal to that process ID, the process ID cannot be reused by the system until the process group lifetime ends.

process lifetime
After a process is created with a *fork*(2) function, it is considered active. Its thread of control and **address space** exist until it terminates. It then enters an inactive state where certain resources may be returned to the system, although some resources, such as the **process ID** are still in use. When another process executes a **wait()**, **wait3()**, or **waitpid()** function (see *wait*(2)) for an inactive process, the remaining resources are returned to the system. The last resource to be returned to the system is the process ID. At this time, the lifetime of the process ends.

program
A sequence of instructions to the computer in the form of binary code (resulting from the compilation and assembly of program source).

prompt

The characters displayed by the **shell** on the terminal indicating that the system is ready for a command. The prompt is usually a dollar sign (**$**) for ordinary users (**%** in the C shell) and a pound sign (**#**) for the superuser, but you can redefine it to be any string by setting the appropriate shell variable (see *sh*(1) and related entries). See also **secondary prompt**.

quit signal

The **SIGQUIT** signal (see *signal*(2). The quit signal is generated by typing the character defined by the teletype handler as your quit signal. (See *stty*(1), *ioctl*(2), and *termio*(7).) The default is the ASCII FS character (ASCII value 28) generated by typing Ctrl-****. This signal usually causes a running program to terminate and generates a file containing the "core image" of the terminated process. The core image is useful for debugging purposes. (Some systems do not support core images, and on those systems no such file is generated.)

radix character

The character that separates the integer part of a number from the fractional part. For example, in American usage, the **radix character** is a decimal point, while in Europe, a comma is used.

raw disk

The name given to a disk for which there exists a **character special file** that allows direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O call.

read-only file system

A characteristic of a **file system** that prevents file system modifications.

real group ID

A positive integer which is assigned to every user on the system. The association of a user and his or her **real group ID** is done in the file **/etc/passwd**. The modifier "real" is used because a user can also have an **effective group ID**. The real group ID can then be mapped to a group name in the file **/etc/group**, although it need not be. Thus, every user is a member of some group (which can be nameless), even if that group has only one member.

Every time a process creates a child process (via *fork*(2)), that process has a real group ID equal to the parent process's real group ID. This is useful for determining file access privileges within the process.

real user ID

A positive integer which is assigned to every user on the system. A real user ID is assigned to every valid **login** name in the file **/etc/passwd**. The modifier "real" is used because a user can also have an **effective user ID** (see **effective user ID**).

Every time a process creates a child process (via *fork*(2)), that process has a real user ID equal to the parent process's real user ID. This is useful for determining file access privileges within the process.

regular expression

A string of zero or more characters that selects text. All the characters contained in the string might be literal, meaning that the regular expression matches itself only; or one or more of the characters might be a **metacharacter**, meaning that a single regular expression could match several literal strings. Regular expressions are most often encountered in text editors (such as *ed*(1), *ex*(1), or *vi*(1)), where searches are performed for a specific piece of text, or in commands that were created to search for a particular string in a file (most notably *grep*(1)). Regular expressions are also encountered in the shell, especially when referring to file names on command lines.

regular file

A type of **file** that is a randomly accessible sequence of bytes, with no further structure imposed by the system. Its size can be extended. A regular file is also called an **ordinary file**.

relative path name

A **path name** that does not begin with a **slash** (**/**). It indicates that a file's location is given relative to your current **working directory**, and that the search begins there (instead of at the **root directory**). For example, **dir1/file2** searches for the directory **dir1** in your current working directory; then **dir1** is searched for the file **file2**.

p

__restrict
A macro that is optionally applied to the function prototype when the application developer directly or indirectly selects C99 conformance. If the user chooses C99 conformance, the **__restrict** macro is changed to the **restrict** keyword. Otherwise, the **__restrict** macro is expanded to an empty string.

root directory
(1) The highest level directory of the hierarchical file system, from which all other files branch. In HP-UX, the **slash** (**/**) character refers to the **root directory**. The root directory is the only directory in the file system that is its own **parent directory**.

(2) Each process has associated with it a concept of a root directory for the purpose of resolving path name searches for those paths beginning with **slash** (**/**). A process's root directory need not be the root directory of the root file system, and can be changed by the *chroot*(1M) command or *chroot*(2) system call. Such a directory appears to the process involved to be its own parent directory.

root volume
The mass storage volume which contains the boot area (which contains the HP-UX kernel) and the **root directory** of the HP-UX file system.

saved group ID
Every process has a saved group ID that retains the process's **effective group ID** from the last successful *exec*(2) or **setresgid()** (see *setresuid*(2)), or from the last superuser call to **setgid()** (see *setuid*(2)) or *setresuid*(2). **setgid()** permits a process to set its effective group ID to this remembered value. Consequently, a process that executes a program with the set-group-ID bit set and with a group ID of 5 (for example) can set its effective group ID to 5 at any time until the program terminates. See *exec*(2), *setuid*(2), **saved user ID**, **effective group ID**, and **set-group-ID bit**. The saved group ID is also known as the **saved set-group-ID**.

saved process group ID
Every process has a saved process group ID that retains the process's group ID from the last successful *exec*(2). See *setpgrp*(2), *termio*(7), and **process group ID**.

saved user ID
Every process has a **saved user ID** that retains the process's **effective user ID** from the last successful *exec*(2) or *setresuid*(2), or from the last superuser call to *setuid*(2). *setuid*(2) permits a process to set its effective user ID to this remembered value. Consequently, a process which executes a program with the set-user-ID bit set and with an owner ID of 5 (for example) can set its effective user ID to 5 at any time until the program terminates. See *exec*(2), *setuid*(2), **saved group ID**, **effective user ID**, and **set-user-ID bit**. The saved user ID is also known as the **saved set-user-ID**.

saved set-group-ID
See **saved group ID**.

saved set-user-ID
See **saved user ID**.

SCCS
See **Source Code Control System**.

Source Code Control System (SCCS)
A set of HP-UX commands that enables you to store changes to an **SCCS file** as separate "units" (called **deltas**). These units, each of which contains one or more textual changes to the file, can then be applied to or excluded from the SCCS file to obtain different versions of the file. The commands that make up SCCS are *admin*(1), *cdc*(1), *delta*(1), *get*(1), *prs*(1), *rmdel*(1), *sact*(1), *sccsdiff*(1), *unget*(1), *val*(1), and *what*(1).

SCCS file
An ordinary text file that has been modified so the **Source Code Control System** (**SCCS**) can be used with it. This modification is done automatically by the *admin*(1) command. See also **delta**.

secondary prompt
One or more characters that the shell prints on the display, indicating that more input is needed. This prompt is not encountered nearly as frequently as the shell's primary prompt (see **prompt**). When it occurs, it is usually caused by an omitted right quote on a string (which confuses the shell), or when you

enter a shell programming language control-flow construct (such as a **for** construct) from the command line. By default, the shell's secondary prompt is the greater-than sign (**>**), but you can re-define it by setting the shell variable **PS2** appropriately in your **.profile** file. (The C shell has no secondary prompt.)

semaphore identifier (semid)

A unique positive integer created by a *semget*(2) system call. Each **semid** has a set of semaphores and a data structure associated with it. The data structure is referred to as **semid_ds** and contains the following members:

```
struct
ipc_perm   sem_perm;    /* operation permission */
ushort     sem_nsems;   /* number of sems in set */
time_t     sem_otime;   /* last operation time */
time_t     sem_ctime;   /* last change time */
                        /* Times measured in secs since */
                        /* 00:00:00 GMT, Jan. 1, 1970 */
```

Semaphore identifiers can be created using *ftok*(3C).

sem_perm is a **ipc_perm** structure that specifies the semaphore operation permission (see below). This structure includes the following members:

```
uid_t    cuid;    /* creator user id */
gid_t    cgid;    /* creator group id */
uid_t    uid;     /* user id */
gid_t    gid;     /* group id */
mode_t   mode;    /* r/a permission */
```

The value of **sem_nsems** is equal to the number of semaphores in the set. Each semaphore in the set is referenced by a positive integer referred to as a **sem_num**. **sem_num** values run sequentially from 0 to the value of sem_nsems minus 1. **sem_otime** is the time of the last *semop*(2) operation, and **sem_ctime** is the time of the last *semctl*(2) operation that changed a member of the above structure.

semaphore operation permissions

In the *semop*(2) and *semctl*(2) system call descriptions, the permission required for an operation is indicated for each operation. Whether a particular process has these permissions for an object is determined by the object's permission mode bits as follows:

00400	Read by user
00200	Alter by user
00060	Read, Alter by group
00006	Read, Alter by others

Read and Alter permissions on a **semid** are granted to a process if one or more of the following are true:

- The process's effective user ID is superuser.
- The process's effective user ID matches **sem_perm.**[**c**]**uid** in the data structure associated with **semid** and the appropriate bit of the "user" portion (0600) of **sem_perm.mode** is set.
- The process's effective user ID does not match **sem_perm.**[**c**]**uid** and the appropriate bit of the "group" portion (060) of **sem_perm.mode** is set.
- The process's effective user ID does not match **sem_perm.**[**c**]**uid** and the process's effective group ID does not match **sem_perm.**[**c**]**gid** and neither of **sem_perm.**[**c**]**gid** is in the process's group access list and the appropriate bit of the "other" portion (06) of **sem_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

semid

See **semaphore identifier**.

session

Each process group is a member of a session. A process is considered to be a member of the session of which its process group is a member. A newly created process joins the session of its creator. A process can alter its session membership (see *setsid*(2)). A session can have multiple process groups (see *setpgid*(2)).

S

session leader
A process that has created a session (see *setsid*(2)).

session lifetime
The period between when a session is created and the end of the lifetime of all process groups that remain as members of the session.

set-group-ID bit
A single bit in the mode of every file in the file system. If a file is executed whose **set-group-ID bit** is set, the **effective group ID** of the process which executed the file is set equal to the **real group ID** of the owner of the file. See also **group**.

set-user-ID bit
A single bit in the mode of every file in the file system. If a file is executed whose **set-user-ID bit** is set, the **effective user ID** of the process that executed the file is set equal to the **real user ID** of the owner of the file.

shared library
An executable file that can be shared between several different programs. Code from a shared library is not linked into the program by *ld*(1), but is instead mapped into the process's address space at run time by the dynamic loader. Shared libraries must contain position-independent code, and are created by *ld*(1). They typically have the file name suffix **.sl**.

shared memory identifier (shmid)
A unique positive integer created by a *shmget*(2) system call. Each **shmid** has a segment of memory (referred to as a shared memory segment) and a data structure associated with it. The data structure is referred to as **shmid_ds** and contains the following members:

```
struct
ipc_perm  shm_perm;    /* operation permission struct */
size_t    shm_segsz;   /* size of segment */
pid_t     shm_cpid;    /* creator pid */
pid_t     shm_lpid;    /* pid of last operation */
shmatt_t  shm_nattch;  /* number of current attaches */
time_t    shm_atime;   /* last attach time */
time_t    shm_dtime;   /* last detach time */
time_t    shm_ctime;   /* last change time */
                       /* Times measured in secs since */
                       /* 00:00:00 GMT, Jan. 1, 1970 */
```

Shared memory identifiers can be created using *ftok*(3C).

shm_perm is a **ipc_perm** structure that specifies the permission for a *shmop*(2) or *shmctl*(2) operation (see below). This structure includes the following members:

S

```
uid_t   cuid;     /* creator user id */
gid_t   cgid;     /* creator group id */
uid_t   uid;      /* user id */
gid_t   gid;      /* group id */
mode_t  mode;     /* r/w permission */
```

shm_segsz specifies the size of the shared memory segment. **shm_cpid** is the process id of the process that created the shared memory identifier. **shm_lpid** is the process id of the last process that performed a *shmop*(2) operation. **shm_nattch** is the number of processes that currently have this segment attached. **shm_atime** is the time of the last **shmat** operation, **shm_dtime** is the time of the last **shmdt** operation, and **shm_ctime** is the time of the last *shmctl*(2) operation that changed one of the members of the above structure.

shared memory operation permissions
In the *shmop*(2) and *shmctl*(2) system call descriptions, the permission required for an operation is indicated for each operation. Whether a particular process has the permission to perform a *shmop*(2) or *shmctl*(2) operation on an object is determined by the object's permission mode bits as follows:

00400 Read by user

00200	Write by user
00060	Read, Write by group
00006	Read, Write by others

Read and Write permissions for a *shmop*(2) or *shmctl*(2) operation on a **shared memory identifier** (**shmid**) are granted to a process if one or more of the following are true:

- The process's effective user ID is superuser.
- The process's effective user ID matches **shm_perm.**[**c**]**uid** in the data structure associated with the **shmid** and the appropriate bit of the "user" portion (0600) of **shm_perm.mode** is set.
- The process's effective user ID does not match **shm_perm.**[**c**]**uid** and either the process's effective group ID matches **shm_perm.**[**c**]**gid** or one of **shm_perm.**[**c**]**gid** is in the process's group access list and the appropriate bit of the "group" portion (060) of **shm_perm.mode** is set.
- The process's effective user ID does not match **shm_perm.**[**c**]**uid** and the process's effective group ID does not match **shm_perm.**[**c**]**gid** and neither of **shm_perm.**[**c**]**gid** is in the process's group access list and the appropriate bit of the "other" portion (06) of **shm_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

shell
A user interface to the HP-UX operating system. A shell often functions as both a command interpreter and an interpretive programming language. A shell is automatically invoked for every user who logs in. See *sh*(1) and its related manual entries plus the tutorials supplied with your system for details.

shell program
See **shell script**.

shell script
A sequence of shell commands and shell programming language constructs stored in a file and invoked as a user command (program). No compilation is needed prior to execution because the shell recognizes the commands and constructs that make up the shell programming language. A shell script is often called a **shell program** or a **command file**. See the *Shells User Guide*.

shmid
See **shared memory identifier**.

signal
A software interrupt sent to a process, informing it of special situations or events. Also, the event itself. See *signal*(2).

single-user state
A condition of the HP-UX operating system in which the system console provides the only communication mechanism between the system and its user. By convention, single-user state is usually specified by *init*(1M) as run-level **S** or **s**. Do not confuse **single-user state**, in which the software is limiting a multiuser system to a single-user communication, with a single-user system, which can never communicate with more than one fixed terminal. See also **multiuser state**.

slash
The literal character **/**. A **path name** consisting of a single slash resolves to the **root directory** of the process. See also **path name resolution**.

solidus
See **slash**.

source code
The fundamental high-level information (program) written in the syntax of a specified computer language. Object (machine-language) code is derived from source code. When dealing with an HP-UX shell command language, **source code** is input to the command language interpreter. The term **shell script** is synonymous with this meaning. When dealing with the C Language, **source code** is input to the *cc*(1) command. **Source code** can also refer to a collection of sources meeting any of the above conditions.

special file
A file associated with an I/O device. Often called a **device file**. Special files are read and written the same as **ordinary files**, but requests to read or write result in activation of the associated device. Due to convention and consistency, these files should always reside in the **/dev** directory. See also **file**.

special system processes
Special system processes are those which are critical to basic system operation. They include: the scheduler, the initialization process (also known as **init**) and the pager.

SS/80
See **CS/80**.

standard error
The destination of error and special messages from a program, intended to be used for diagnostic messages. The standard error output is often called **stderr**, and is automatically opened for writing on file descriptor 2 for every command invoked. By default, the user's terminal is the destination of all data written to standard error, but it can be redirected elsewhere. Unlike standard input and standard output, which are never used for data transfer in the "wrong" direction, standard error is occasionally read. This is not recommended practice, since I/O redirection is likely to break a program doing this.

standard input
The source of input data for a program. The standard input file is often called **stdin**, and is automatically opened for reading on file descriptor 0 for every command invoked. By default, the user's terminal is the source of all data read from standard input, but it can be redirected from another source.

standard output
The destination of output data from a program. The standard output file is often called **stdout**, and is automatically opened for writing on file descriptor 1 for every command invoked. By default, the user's terminal is the destination of all data written to standard output, but it can be redirected elsewhere.

stderr
See **standard error**.

stdin
See **standard input**.

stdout
See **standard output**.

stream
A term most often used in conjunction with the standard I/O library routines documented in Section 3 of this manual. A stream is simply a file pointer (declared as **FILE *stream**) returned by the *fopen*(3S) library routines. It may or may not have buffering associated with it (by default, buffering is assigned, but this can be modified with *setbuf*(3S)).

sticky bit
A single bit in the mode of every file in the file system. The sticky bit has no significance if it is set on a **regular file**.

If set on a directory, the files in that directory can be removed or renamed only by the owner of the file, the owner of the directory containing the file, or superuser. See also *chmod*(2), *rename*(2), *rmdir*(2), and *unlink*(2).

subdirectory
A directory that is one or more levels lower in the file system hierarchy than a given directory. Sometimes called a **subordinate directory**.

subordinate directory
See **subdirectory**.

Subset 1980
See **CS/80**.

superblock
A block on each file system's mass storage medium which describes the file system. The contents of the superblock vary between implementations. Refer to the system administrator manuals supplied with your system for details.

superuser
The HP-UX system administrator. This user has access to all files, and can perform privileged operations. **superuser** has a **real user ID** and **effective user ID** of 0, and, by convention, the user name of **root**.

superior directory
See **parent directory**.

supplementary group ID
A process has up to **sysconf(_SC_NGROUPS_MAX)** supplementary group IDs used in determining file access permissions, in addition to the effective group ID. The supplementary group IDs of a process are set to the supplementary group IDs of the parent process when the process is created. Note that the value returned from **sysconf(_SC_NGROUPS_MAX)** may be larger than the value of **NGROUPS_MAX** found in **<limits.h>** on certain HP-UX systems.

symbolic link
A type of file that indirectly refers to a path name. See *symlink*(4).

system
The HP-UX operating system. See also **kernel**.

system asynchronous I/O
A method of performing I/O whereby a process informs a driver or subsystem that it wants to know when data has arrived or when it is possible to perform a write request. The driver or subsystem maintains a set of buffers through which the process performs I/O. See *ioctl*(2), *read*(2), *select*(2), and *write*(2) for more information.

system call
An HP-UX operating system kernel function available to the user through a high-level language (such as FORTRAN, Pascal, or C). Also called an "intrinsic" or a "system intrinsic." The available system calls are documented in Section 2 of the *HP-UX Reference*.

system console
A keyboard and display (or terminal) given a unique status by HP-UX and associated with the special file **/dev/console**. All boot ROM error messages, HP-UX system error messages, and certain system status messages are sent to the system console. Under certain conditions (such as the single-user state), the system console provides the only mechanism for communicating with HP-UX. See the System Administrator manuals and user guides provided with your system for details on configuration and use of the system console.

S

system process
A **system process** is a process that runs on behalf of the system. It may have special implementation-defined characteristics.

terminal
A **character special file** that obeys the specifications of *termio*(7).

terminal affiliation
The process by which a process group leader establishes an association between itself and a particular terminal. A terminal becomes affiliated with a process group leader (and subsequently all processes created by the process group leader, see **terminal group**) whenever the process group leader executes (either directly or indirectly) an *open*(2) or *creat*(2) system call to open a terminal. Then, *if* the process which is executing *open*(2) or *creat*(2) is a process group leader, and *if* that process group leader is not yet affiliated with a terminal, and *if* the terminal being opened is not yet affiliated with a process group, the affiliation is established (however, see *open*(2) description of **O_NOCTTY**).

An affiliated terminal keeps track of its process group affiliation by storing the process group's process group ID in an internal structure.

Two benefits are realized by terminal affiliation. First, all signals sent from the terminal are sent to all processes in the terminal group. Second, all processes in the terminal group can perform I/O to/from the generic terminal driver **/dev/tty**, which automatically selects the affiliated terminal.

Terminal affiliation is broken with a terminal group when the process group leader terminates, after which the hangup signal is sent to all processes remaining in the process group. Also, if a process (which is not a process group leader) in the terminal group becomes a process group leader via the *setpgrp*(2) system call, its terminal affiliation is broken.

See **process group**, **process group leader**, **terminal group**, and *setpgrp*(2).

terminal device
See **terminal**.

text file
A file that contains characters organized into one or more lines. The lines cannot contain NUL characters, and none can exceed **LINE_MAX** bytes in length including the terminating newline character. Although neither the kernel nor the C language implementation distinguishes between text files and binary files (see ANSI C Standard X3-159-19xx), many utilities behave predictably only when operating on text files.

tty
Originally, an abbreviation for teletypewriter; now, generally, a **terminal**.

upshifting
The conversion of a lowercase character to its uppercase representation.

user ID
Each system user is identified by an integer known as a **user ID**, which is in the range of zero to **UID_MAX**, inclusive. Depending on how the user is identified with a process, a **user ID** value is referred to as a **real user ID**, an **effective user ID**, or a **saved user ID**.

UTC
See **Epoch**.

utility
An executable file, which might contain executable object code (that is, a **program**), or a list of **commands** to execute in a given order (that is, a **shell script**). You can write your own utilities, either as executable programs or shell scripts (which are written in the shell programming language).

volume number
Part of an address used for devices. A number whose meaning is software- and device-dependent, but which is often used to specify a particular volume on a multivolume disk drive. See the System Administrator manuals supplied with your system for details.

whitespace
One or more characters which, when displayed, cause a movement of the cursor or print head, but do not result in the display of any visible graphic. The whitespace characters in the ASCII code set are space, tab, newline, form feed, carriage return, and vertical tab. A particular command or routine might interpret some, but not necessarily all, whitespace characters as delimiters for fields, words, or command options.

working directory
Each process has associated with it the concept of a current working directory. For a shell, this appears as the directory in which you currently "reside". This is the directory in which relative path name (that is, a path name that does not begin with **/**) searches begin. It is sometimes referred to as the **current directory**, or the **current working directory**.

zombie process
The name given to a process which terminates for any reason, but whose parent process has not yet waited for it to terminate (via *wait*(2)). The process which terminated continues to occupy a slot in the process table until its parent process waits for it. Because it has terminated, however, there is no other space allocated to it either in user or kernel space. It is therefore a relatively harmless occurrence which will rectify itself the next time its parent process waits. The *ps*(1) command lists zombie processes as **defunct**.

SEE ALSO

introduction(9).

NAME

introduction - HP-UX operating system and HP-UX Reference

INTRODUCTION

HP-UX is the Hewlett-Packard Company's implementation of a UNIX® operating system that is compatible with various industry standards. It is based on the System V Release 4 operating system (SVR4) and includes important features from the Fourth Berkeley Software Distribution (4BSD).

Improvements include enhanced capabilities and other features, developed by HP to make HP-UX a very powerful, useful, and reliable operating system, capable of supporting a wide range of applications ranging from simple text processing to sophisticated engineering graphics and design. It can readily be used to control instruments and other peripheral devices. Real-time capabilities further expand the flexibility of HP-UX as a powerful tool for solving tough problems in design, manufacturing, business, and other areas where responsiveness and performance are important.

Extensive international language support enables HP-UX to interact with users in any of dozens of human languages. HP-UX interfaces easily with local area networks and resource-sharing facilities. By using industry-standard protocols, HP-UX provides flexible interaction with other computers and operating systems. Optional software products extend HP-UX capabilities into a broad range of specialized needs.

The *HP-UX Reference* is not a learning tool for beginners. It is primarily a reference tool that is most useful for experienced users of UNIX or UNIX-like systems. If you are not already familiar with UNIX or HP-UX, refer to the series of Beginner's Guides, tutorial manuals, and other learning documents supplied with your system or available separately. System implementation and maintenance details are explained in the *HP-UX System Administrator's Guide*.

OTHER MANPAGES

This introduction and the section *intro* manpages describe the "core" manpages that are delivered with HP-UX. Other manpages may be delivered separately with optional HP-UX and third-party software and may reside in the same directories as the core manpages, or in other directories.

MANPAGE ORGANIZATION

The contents of the *HP-UX Reference* and its on-line counterpart are a number of independent entries called **manpages**. These are also called **manual entries** or **reference pages**.

For convenient reference, the manpages are divided into eight specialized sections. The printed manual also has a table of contents for each volume and a composite index.

Each manpage consists of one or more printed pages, with the manpage name and section number printed in the upper corners. Manpages are arranged alphabetically within each section of the reference, except for the *intro* page at the beginning of each section. Manpages are referred to by name and section number, in the form *pagename*(*section*).

The manpages are available on-line through the **man** command if the manpages are present on the system. Refer to the *man*(1) manpage in Section 1 for more information.

Each page in the printed manual has two page numbers, printed at the bottom of the page. The center page number starts over with page 1 at the beginning of each new manpage; it is placed between two dashes in normal typeface. The number printed at the outside corner on each page is the sequence number of the page within the volume. Users usually locate manpages by the alphabetic headings at the top of the page as when reading a dictionary.

Some manpages describe two or more commands or routines. In such cases, the manpage is usually named for the first command or function that appears in the NAME section. Occasionally, a manpage name appears as a group descriptor in the NAME section. In such instances, the name describes the commands or functions in more general terms. For example, the *acct*(1M) manpage with group descriptor **acct:** describes the **acctdisk**, **acctdusg**, **accton**, and other commands, while the *string*(3C) manpage with group descriptor **string:** describes many character string functions.

SECTIONS OF THE HP-UX REFERENCE

The *HP-UX Reference* contains the following sections:

Volume Table of Contents (Printed Volumes)

A complete listing of all manpages in the order they appear in each section, as well as alphabetically intermixed lists of all command, function, and feature names that are different from the manpage where they appear.

Section 1: User Commands

Programs that are usually invoked directly by users or from command language procedures (scripts).

Section 1M: System Administration Commands

Commands used for system installation and maintenance, including boot processes, crash recovery, system integrity testing, and other needs. Most commands in this section require the superuser privilege.

Section 2: System Calls

Entries into the HP-UX kernel, including the C-language interface. These topics are primarily of interest to programmers.

Section 3: Library Functions

Available subroutines that reside (in binary form) in various system libraries. These topics are primarily of interest to programmers.

Section 4: File Formats

The structure of various types of files, such as header files, primarily of interest to administrators and programmers. For example, the link editor output file format is described in *a.out*(4). Files that are used only by a single command (such as intermediate files used by assemblers) are not described. C-language declarations corresponding to the formats in Section 4 can be found in the directories **/usr/include** and **/usr/include/sys**.

Section 5: Miscellaneous Topics

A variety of information, such as descriptions of character sets, macro packages, and kernel tunables.

Section 6 (Unused)

This section was traditionally used for games. None are shipped with HP-UX.

Section 7: Device Special Files

The characteristics of device special files (DSF) that provide the link between HP-UX and system I/O devices. The names for each topic usually refer to the type of I/O device rather than to the names of individual special files.

Section 8: System Administration Commands

Some UNIX and Linux vendors put system administration commands here. Some third party vendors install commands in this section in HP-UX.

Section 9: General Information

General introductions (such as this) and a glossary of terms used in the HP-UX environment.

This section is also used by the Driver Development Kit to store its function and structure manpages, using the section numbers 9E, 9F, and 9S.

Composite Index (Printed Manual)

An alphabetical listing of keywords and topics based on the NAME section near the beginning of each manpage as well as other information, cross-referenced to manpage names and sections. The index also contains references to built-in features in the various command interpreters ("shells").

MANPAGE FORMATS

All manpages follow an established section heading format, but not all section headings are included in each manpage. A few manpages have self-explanatory specialized headings.

NAME

Gives the names of the commands, functions, or features and briefly states the purpose.

SYNOPSIS

Summarizes the syntax of the command or program entity. A few conventions are used:

`Constant-width` characters indicate literal characters that should be entered exactly as they appear. These characters appear in bold in the online manpages.

Italic strings represent variable elements that should be replaced with appropriate values.

Roman square brackets ([]) indicate that the contents are optional.

Roman braces ({}) indicate a required element, usually in a choice.

Ellipses (...) indicate that the previous element and its preceding whitespace (if any) can be repeated.

Note: An argument beginning with a dash (**-**), a plus sign (**+**), or an equal sign (**=**) is often defined as a command option, even if it appears in a position where a file name could appear. Therefore, it is unwise to have files names that begin with **-**, **+**, or **=**.

Optional subsections can include the following:

Parameters For functions, a description of the parameters in the preceding syntax.

Structure Members For structures, a description of the structure elements in the preceding syntax.

Remarks Information about special software or hardware requirements.

DESCRIPTION

Discusses the function and behavior of each entry.

Optional subsections can include the following:

Options For commands, a description of the switch arguments.

Operands For commands, a description of the nonswitch arguments and keywords.

Access Control Lists

Multithread Usage

Security Restrictions Information on restrictions and privileges required to use the item.

EXTERNAL INFLUENCES

Information on what external factors, such as environment variables, may affect system behavior.

Optional subsections can include the following:

Environment Variables The effect of language-related and other environment variables on system behavior,

International Code Set Support
Whether there is support for single- and multibyte characters,

NETWORKING FEATURES

Information under this heading is applicable only if you are using the network feature described there.

Optional subsections can include the following:

NFS Information on the network file system.

RETURN VALUE

Describes the values returned by function calls or in the return code (**$?**) by commands.

DIAGNOSTICS

For commands, the diagnostic information that may be produced. Self-explanatory messages are not listed.

Optional subsections can include the following:

Errors

Warnings

ERRORS

For functions, the function error values (set in **errno**) and their corresponding error conditions.

EXAMPLES

Examples of typical usage.

WARNINGS

Potential problems and deficiencies.

DEPENDENCIES

Variations in HP-UX operation that are related to the use of specific hardware, software, or combinations of hardware and software.

AUTHOR

Indicates the principal developer of the software documented by the manpage. Unless noted otherwise, the source of an entry is System V.

FILES

The file names that are used or affected by the program or command.

SEE ALSO

Provides references to related manpages and other documentation.

STANDARDS CONFORMANCE

For each command or subroutine entry point addressed by one or more of the following industry standards, the standard specifications to which that HP-UX component conforms.

The various standards are:

AES	OSF Application Environment Specification
ANSI C	ANSI X3.159-1989
FIPS 151-1	Federal Information Processing Standard 151-1 (National Institute of Standards and Technology)
FIPS 151-2	Federal Information Processing Standard 151-2 (National Institute of Standards and Technology)
POSIX.1	IEEE Standard 1003.1-1988 (IEEE Computer Society) (Portable Operating System Interface for Computer Environments)
POSIX.2	IEEE Standard 1003.2-1990 (IEEE Computer Society) (Portable Operating System Interface for Computer Environments)
POSIX.4	IEEE Standard 1003.1b-1993 (IEEE Computer Society) (Portable Operating System Interface for Computer Environments)
SVID2	System V Interface Definition Issue 2
SVID3	System V Interface Definition Issue 3
XPG2	X/Open Portability Guide Issue 2 (X/Open, Ltd.)
XPG3	X/Open Portability Guide Issue 3 (X/Open, Ltd.)
XPG4	X/Open Portability Guide Issue 4 (X/Open, Ltd.)
XPG4.2	X/Open Portability Guide Issue 4 Version 2 (X/Open, Ltd.)

GETTING STARTED WITH HP-UX

This is a very brief overview of how to use the HP-UX system: how to log in and log out, how to communicate through your machine, and how to run a program.

HP-UX uses **control characters** to perform certain functions. Control characters are generally shown in the form `^`*x*, such as **`^D`** for Control-D. Hold down the **Control** (**Ctrl**) key while you press the character key.

Note: The key names **Enter** and **Return** refer to the same key.

Logging In

To log in you must have a valid user name and password, which can be obtained from your system administrator.

When a connection has been established, the system displays **`login:`** on your terminal. Type your user name and press the **Enter** key. Enter your password (it is not echoed by the system) and press **Enter**.

A list of copyright notices and a message-of-the-day may greet you before the first prompt.

It is important that you type your login name with lowercase letters, if possible. If you type uppercase letters, HP-UX assumes that your terminal cannot generate lowercase letters, and treats subsequent uppercase input as lowercase.

When you log in successfully, the system starts your login shell. The default is the POSIX shell, **`/usr/bin/sh`**. The POSIX shell (and its predecessors, the Korn and Bourne shells) use **`$`** as the default prompt for users. The C shell uses **%**. All the shells use **#** as the default superuser prompt.

See *login*(1) for more on login, *passwd*(1) to change your password, *chsh*(1) to change your login shell.

Logging Out

You can log out of the shells by typing an **exit** command or the **eof** (end-of-file) character (see the *Special Interactive Characters* subsection below). The shell terminates and the **login:** prompt appears again. (If you are using the C, Korn, or POSIX shells, respectively, see *csh*(1), *ksh*(1), or *sh-posix*(1) for information about the **ignoreeof** special command.)

How to Communicate Through Your Terminal

HP-UX gathers keyboard input characters and saves them in a buffer. The accumulated characters are not passed to the shell or other program until you type **Enter**.

HP-UX terminal input/output is full-duplex. It has full read-ahead, which means that you can type at any time, even while a program is printing on your display or terminal. Of course, if you type during output, the output display will have the input characters interspersed in it. However, whatever you type will be saved and interpreted in the correct sequence. There is a limit to the amount of read-ahead, but it is generous and not likely to be exceeded unless the system is severely overloaded or operating abnormally. When the read-ahead limit is exceeded, the system throws away *all* the saved characters.

The *stty*(1) manpage tells you how to describe the characteristics of your terminal to the system. The *profile*(4) manpage explains how to accomplish this task automatically every time you log in.

Special Interactive Characters

A number of special characters are used to control the input and output of your terminal. These characters have defaults and can be redefined with the **stty** command (see *stty*(1)). Definitions of the **stty** names are in *termio*(7) and *termiox*(7).

Note: The system administrator can modify the system login defaults by changing the characteristics of the **/dev/ttyconf** device file with the **stty** command.

stty Name	System Default At Login Character (ASCII Name; Key Names)	Common User Redefinition
eof	**^D** (EOT)	
erase	**#**	**^H** (BS; Backspace)
kill	**@**	**^U** (NAK), **^X** (CAN)
intr	**^?** (DEL; Delete, Rub, Rubout)	**^C** (ETX)
quit	**^** (FS)	
start	**^Q** (DC1; X-ON)	
stop	**^S** (DC3; X-OFF)	

The **eof** character terminates "file" input from the terminal, as read by programs and scripts. By extension, **eof** can also terminate the shell (see the *Logging Out* subsection above).

The **erase** character erases the last character typed. Successive uses of **erase** will erase characters back to, but not beyond, the beginning of the input line.

The **kill** character deletes all characters typed before it on a terminal input line.

The **intr** character generates an interrupt signal that bypasses the input buffer. This signal generally causes whatever program you are running to terminate. It can be used to stop a long printout that you don't want. However, programs can arrange either to ignore this signal altogether, or to be notified when it happens (instead of being terminated). For example, the **vi** editor catches interrupts and stops what it is doing, instead of terminating, so that an interrupt can be used to halt an editing operation without losing the file being edited.

The **quit** character generates a quit signal that bypasses the input buffer and most program traps and causes a running program to terminate. It can cause a core dump in the current directory.

The **stop** character can be used to pause output to the terminal. It is commonly used on video terminals to suspend output to the display while you read what is already being displayed. You can then resume output by typing the **start** character. When **stop** and **start** are used to suspend or resume output, they bypass the keyboard command-line buffer and are not passed to the program. However, any other characters typed on the keyboard are saved and used as input later in the program.

The **eof**, **erase**, and **kill** characters can be used as normal text characters if you escape them with a preceding ****, as in **\^D**. Therefore, to erase a ****, you need two **erase**s.

The **intr**, **quit**, **start**, and **stop** characters cannot be escaped on the input line.

End-of-Line and Tab Characters

Besides adapting to the speed of the terminal, HP-UX tries to be intelligent as to whether you have a terminal with a newline (line-feed) key, or whether it must be simulated with a return/line-feed character pair. In the latter case, all incoming return characters are changed to line-feed characters (the standard line delimiter), and a return/line-feed pair is echoed to the terminal. If you get into the wrong mode, use the **stty** command to correct it (see *stty*(1)).

Tab characters are used freely in HP-UX source programs. If your terminal does not have the tab function, you can arrange to have tab characters changed into spaces during output, and echoed as spaces during input. The **stty** command sets or resets this mode. By default, the system assumes that tabs are set every eight character positions. The **tabs** command (see *tabs*(1)) can set tab stops on your terminal, if the terminal supports tabs.

How to Run a Program

When you have successfully logged into HP-UX, the shell monitors input from your terminal. The shell accepts typed lines from the terminal, splits them into command names and arguments, then executes the command. The command can be the name of a shell built-in, an executable script of commands, or an executable program. There is nothing special about system-provided commands, except that they are kept in directories where the shell can find them. You can also keep commands in your own directories and arrange for the shell to find them there.

The command name is the first word on an input line to the shell; the command and its arguments are separated from one another by blanks (one or more space and/or tab characters).

When a program terminates, the shell ordinarily regains control and prompts you to indicate that it is ready for another command. The shell has many other capabilities, which are described in detail in the appropriate manpages: *sh-posix*(1) for the POSIX shell, *ksh*(1) for the Korn shell, or *csh*(1) for the C shell.

The Current Directory

HP-UX has a file system arranged in a hierarchy of directories. When the system administrator gave you a user name, he or she also created a directory for you (ordinarily with the same name as your user name, and known as your *login* or *home* directory). When you log in, that directory becomes your *current* or *working* directory, and any file name you type is assumed to be in that directory by default. Because you are the owner of this directory, you have full permission to read, write, alter, or destroy its contents. The permissions you have for other directories and files will have been granted or denied to you by their respective owners, or by the system administrator. To change the current working directory use the **cd** command (see *cd*(1)).

Path Names

To refer to files not in the current directory, you must use a path name. Full (absolute) path names begin with **/**, which is the name of the *root* directory of the whole file system. After the slash comes the name of each directory containing the next subdirectory (followed by a **/**), until finally the file name is reached (for example, **/usr/ae/filex** refers to file **filex** in directory **ae**, while **ae** is itself a subdirectory of **usr**; **usr** is a subdirectory of the root directory). See *glossary*(9) for a formal definition of **path name**.

If your current directory contains subdirectories, the path names of files in them begin with the name of the corresponding subdirectory (without a prefixed **/**). Generally, a path name can be used anywhere a file name is required.

Important commands that modify the contents of directories are **cp**, **mv**, and **rm** which respectively copy, move (that is, rename, relocate, or both), and remove files. To determine the status of files or the contents of directories, use the **ls** command. Use **mkdir** to make directories, **rmdir** to destroy them, and **mv** to rename them. See *cp*(1), *ls*(1), *mkdir*(1), *mv*(1), *rm*(1), and *rmdir*(1).

Writing a Program

To enter the text of a source program into an HP-UX file, use a text editing program such as **vi**, **ex**, or **ed** (see *vi*(1), *ex*(1), and *ed*(1)). The three principal languages available under HP-UX are C (see *cc_bundled*(1) and *cc*(1)), FORTRAN (see *f77*(1)), and aC++ (see *aCC*(1)). After the program text has been entered with the editor and written into a file (whose name has the appropriate suffix), you can give the name of that file to the appropriate language processor as an argument. Normally, the output of the language processor will be left in a file named **a.out** in the current directory. Since the results of a subsequent compilation may also be placed in **a.out**, thus overwriting the current output, you may want to use **mv** to give the output a unique name. If the program is written in assembly language, you will probably need to link library subroutines with it (see *ld*(1)). FORTRAN, C, and aC++ call the linker automatically.

When you have gone through this entire process without encountering any diagnostics, the resulting program can be run by giving its name to the shell in response to the prompt.

Your programs can receive arguments from the command line just as system programs do by using the *argc* and *argv* parameters. For more information, see your language's *Programmer's Guide*.

Text Processing

Almost all text is entered through a text editor. The editor preferred above all others provided with HP-UX is the **vi** editor. For batch-processing text files, the **sed** editor is very efficient. The **ex** editor is useful for handling certain situations while using **vi** but most other editors are rarely used except in various scripts.

The following editors are the same program masquerading under various names: **vi**, **view**, and **vedit** (see *vi*(1)) and **ex** and **edit** (see *ex*(1)). For information about the **sed** stream editor, see *sed*(1). The **ed** line editor is described in *ed*(1).

The commands most often used to display text on a terminal are **cat**, **more**, and **pr**. See *cat*(1), *more*(1), and *pr*(1). The **cat** command simply copies ASCII text to the terminal, with no processing at all. The **more** command displays text on the terminal a screenful at a time, pausing for an acknowledgement from the user before continuing. The **pr** command paginates text, supplies headings, and has a facility for multicolumn output. **pr** is most commonly used in conjunction with the **lp** command (see *lp*(1)) to pipe formatted text to a line printer.

Interuser Communication

Certain commands provide interuser communication. Even if you do not plan to use them, it could be beneficial to learn about them, because someone else may direct them toward you. To communicate with another user that is currently logged in, you can use **write** to transfer text directly to that user's terminal display (if permission to do so has been granted by the other user). Otherwise, **elm**, **mailx**, or **mail** (in order of ease of use) can send a message to another user's mailbox. The user is then informed by HP-UX that mail has arrived (if currently logged in) or mail is present (when the user next logs in). Refer to *elm*(1), *mail*(1), *mailx*(1), and *write*(1) for explanations of how these commands are used.

ACKNOWLEDGEMENTS

UNIX is a registered trademark of The Open Group.

SEE ALSO

cat(1), cc_bundled(1), cd(1), chsh(1), cp(1), csh(1), ed(1), ex(1), ksh(1), ld(1), login(1), lp(1), ls(1), mail(1), mailx(1), man(1), mkdir(1), more(1), mv(1), passwd(1), pr(1), rm(1), rmdir(1), sed(1), sh(1), sh-posix(1), stty(1), tabs(1), vi(1), write(1), a.out(4), profile(4), glossary(9).

The HP Technical Documentation website at: **http://docs.hp.com**.

Index

All Volumes

Index

All Volumes

Index
All Volumes

Description | **Entry Name(Section)**

Description — **Entry Name(Section)**

Description **Entry Name(Section)**

Description — **Entry Name(Section)**

Description **Entry Name(Section)**

Description **Entry Name(Section)**

Description — **Entry Name(Section)**

Description **Entry Name(Section)**

Description **Entry Name(Section)**

Description — **Entry Name(Section)**

Description **Entry Name(Section)**

Description **Entry Name(Section)**

Description | Entry Name(Section)

Description | **Entry Name(Section)**

Description **Entry Name(Section)**

Description **Entry Name(Section)**

Description **Entry Name(Section)**

Description **Entry Name(Section)**

Description **Entry Name(Section)**

Description **Entry Name(Section)**

Description — **Entry Name(Section)**

Description **Entry Name(Section)**

Notes