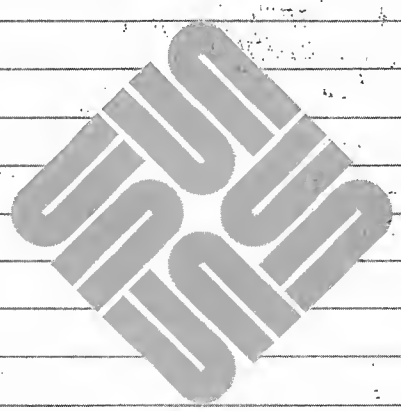
sun microsystems®

SunOS Reference Manual



Part Number: 800-3827-10
Revision A of 27 March, 1990

Thc Sun logo, Sun Microsystems, Sun Workstation, NFS, and TOPS are registered trademarks of Sun Microsystems, Inc.

Sun, Sun-2, Sun-3, Sun-4, Sun386*i*, SPARCstation, SPARCserver, NeWS, NSE, OpenWindows, SPARC, SunInstall, SunLink, SunNet, SunOS, SunPro, and SunView are trademarks of Sun Microsystems, Inc.

UNIX is a registered trademark of AT&T; OPEN LOOK is a trademark of AT&T.

All other products or services mentioned in this document are identified by the trademarks or service marks of their respective companies or organizations, and Sun Microsystems, Inc. disclaims any responsibility for specifying which marks are owned by which companies or organizations.

This logo is a trademark of the X/Open Company Limited in the UK and other countries, and its use is licensed to Sun Microsystems, Inc. The use of this logo certifies SunOS 4.1 conformance with X/Open Portability Guide Issue 2 (XPG 2).

Copyright © 1987, 1988, 1989, 1990 Sun Microsystems, Inc. – Printed in U.S.A.

All rights reserved. No part of this work covered by copyright hereon may be reproduced in any form or by any means – graphic, electronic, or mechanical – including photocopying, recording, taping, or storage in an information retrieval system, without the prior written permission of the copyright owner.

Restricted rights legend: use, duplication, or disclosure by the U.S. government is subject to restrictions set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 52.227-7013 and in similar clauses in the FAR and NASA FAR Supplement.

The Sun Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees.

This product is protected by one or more of the following U.S. patents: 4,777,485 4,688,190 4,527,232 4,745,407 4,679,014 4,435,792 4,719,569 4,550,368 in addition to foreign patents and applications pending.

This software and documentation is based in part on the Fourth Berkeley Software Distribution under license from the Regents of the University of California. We acknowledge the following individuals and institutions for their role in its development: The Regents of the University of California, the Electrical Engineering and Computer Sciences Department at the Berkeley Campus of the University of California, and Other Contributors.

NAME

intro – introduction to device drivers, protocols, and network interfaces

DESCRIPTION

This section describes device drivers, high-speed network interfaces, and protocols available under SunOS. The system provides drivers for a variety of hardware devices, such as disks, magnetic tapes, serial communication lines, mice and frame buffers, as well as virtual devices such as pseudo-terminals and windows. SunOS provides hardware support and a network interface for the 10-Megabit Ethernet, along with interfaces for the IP protocol family and a STREAMS-based Network Interface Tap (NIT) facility.

In addition to describing device drivers that are supported by the 4.3BSD operating system, this section contains subsections that describe:

- SunOS-specific device drivers, under '4S'.
- Protocol families, under '4F'.
- Protocols and raw interfaces, under '4P'.
- STREAMS modules, under '4M'.
- Network interfaces, under '4N'.

Configuration

The SunOS kernel can be configured to include or omit many of the device drivers described in this section. The CONFIG section of the manual page gives the line(s) to include in the kernel configuration file for each machine architecture on which a device is supported. If no specific architectures are indicated, the configuration syntax applies to all Sun systems.

The GENERIC kernel is the default configuration for SunOS. It contains all of the optional drivers for a given machine architecture. See **config**(8), for details on configuring a new SunOS kernel.

The manual page for a device driver may also include a DIAGNOSTICS section, listing error messages that the driver might produce. Normally, these messages are logged to the appropriate system log using the kernel's standard message-buffering mechanism (see **syslogd**(8)); they may also appear on the system console.

Ioctls

Various special functions, such as querying or altering the operating characteristics of a device, are performed by supplying appropriate parameters to the **ioctl**(2) system call. These parameters are often referred to as "ioctls." Ioctls for a specific device are presented in the manual page for that device. Ioctls that pertain to a class of devices are listed in a manual page with a name that suggests the class of device, and ending in 'io', such as **mtio**(4) for magnetic tape devices, or **dkio**(4S) for disk controllers. In addition, some ioctls operate directly on higher-level objects such as files, terminals, sockets, and streams:

- Ioctls that operate directly on files, file descriptors, and sockets are described in **filio**(4). Note: the **fcntl**(2V) system call is the primary method for operating on file descriptors as such, rather than on the underlying files. Also note that the **setsockopt** system call (see **getsockopt**(2)) is the primary method for operating on sockets as such, rather than on the underlying protocol or network interface. Ioctls for a specific network interface are documented in the manual page for that interface.
- Ioctls for terminals, including pseudo-terminals, are described in **termio**(4). This manual page includes information about both the BSD **termios** structure, as well as the System V **termio** structure.
- Ioctls for STREAMS are described in **streamio**(4).

Devices Always Present

Device drivers present in every kernel include:

- The paging device; see **drum**(4).
- Drivers for accessing physical, virtual, and I/O space in memory; see **mem**(4S).
- The data sink; see **null**(4).

Terminals and Serial Communications Devices

Serial communication lines are normally supported by the terminal driver; see **tty**(4). This driver manages serial lines provided by communications drivers, such as those described in **mti**(4S) and **zs**(4S). The terminal driver also handles serial lines provided by virtual terminals, such as the Sun console monitor described in **console**(4S), and true pseudo-terminals, described in **pty**(4).

Disk Devices

Drivers for the following disk controllers provide standard block and raw interfaces under SunOS;

- SCSI controllers, in **sd**(4S),
- Xylogics 450 and 451 SMD controllers, in **xy**(4S),
- Xylogics 7053 SMD controllers, in **xd**(4S).

Ioctls to query or set a disk's geometry and partitioning are described in **dkio**(4S).

Magnetic Tape Devices

Magnetic tape devices supported by SunOS include those described in **ar**(4S), **tm**(4S), **st**(4S), and **xt**(4S). Ioctls for all tape-device drivers are described in **mtio**(4S).

Frame Buffers

Frame buffer devices include color frame buffers described in the **cg***(4S) manual pages, monochrome frame buffers described in the **bw***(4S) manual pages, graphics processor interfaces described in the **gp***(4S) manual pages, and an indirect device for the console frame buffer described in **fb**(4S). Ioctls for all frame-buffer devices are described in **fbio**(4S).

Miscellaneous Devices

Miscellaneous devices include the console keyboard described in **kbd**(4S), the console mouse described in **mouse**(4S), window devices described in **win**(4S), and the DES encryption-chip interface described in **des**(4S).

Network-Interface Devices

SunOS supports the 10-Megabit Ethernet as its primary network interface; see **ie**(4S) and **le**(4S) for details. However, a software loopback interface, **lo**(4) is also supported. General properties of these network interfaces are described in **if**(4N), along with the ioctls that operate on them.

Support for network routing is described in **routing**(4N).

Protocols and Protocol Families

SunOS supports both socket-based and STREAMS-based network communications. The Internet protocol family, described in **inet**(4F), is the primary protocol family primary supported by SunOS, although the system can support a number of others. The raw interface provides low-level services, such as packet fragmentation and reassembly, routing, addressing, and basic transport for socket-based implementations. Facilities for communicating using an Internet-family protocol are generally accessed by specifying the AF_INET address family when binding a socket; see **socket**(2) for details.

Major protocols in the Internet family include:

- The Internet Protocol (IP) itself, which supports the universal datagram format, as described in **ip**(4P). This is the default protocol for SOCK_RAW type sockets within the AF_INET domain.
- The Transmission Control Protocol (TCP); see **tcp**(4P). This is the default protocol for SOCK_STREAM type sockets.
- The User Datagram Protocol (UDP); see **udp**(4P). This is the default protocol for SOCK_DGRAM type sockets.
- The Address Resolution Protocol (ARP); see **arp**(4P).
- The Internet Control Message Protocol (ICMP); see **icmp**(4P).

The Network Interface Tap (NIT) protocol, described in **nit**(4P), is a STREAMS-based facility for accessing the network at the link level.

SEE ALSO

fcntl(2V), **getsockopt**(2), **ioctl**(2), **socket**(2), **ar**(4S), **arp**(4P), **dkio**(4S), **drum**(4), **fb**(4S), **fbio**(4S), **filio**(4), **icmp**(4P), **if**(4N), **inet**(4F), **ip**(4P), **kbd**(4S), **le**(4S), **lo**(4), **mem**(4S), **mti**(4S), **mtio**(4), **nit**(4P), **null**(4), **pty**(4), **routing**(4N), **sd**(4S), **st**(4S) **streamio**(4), **tcp**(4P), **termio**(4), **tm**(4S), **tty**(4), **udp**(4P), **win**(4S), **xd**(4S), **xy**(4S), **zs**(4S)

LIST OF DEVICES, INTERFACES AND PROTOCOLS

Name	Appears on Page	Description
alm	**mcp**(4S)	ALM-2 Asynchronous Line Multiplexer
ar	**ar**(4S)	Archive 1/4 inch Streaming Tape Drive
arp	**arp**(4P)	Address Resolution Protocol
atbus	**mem**(4S)	main memory and bus I/O space
audio	**audio**(4)	telephone quality audio device
bwtwo	**bwtwo**(4S)	black and white memory frame buffer
cdromio	**cdromio**(4S)	CDROM control operations
cgeight	**cgeight**(4S)	24-bit color memory frame buffer
cgfour	**cgfour**(4S)	Sun-3 color memory frame buffer
cgnine	**cgnine**(4S)	24-bit VME color memory frame buffer
cgsix	**cgsix**(4S)	accelerated 8-bit color frame buffer
cgthree	**cgthree**(4S)	8-bit color memory frame buffer
cgtwo	**cgtwo**(4S)	color graphics interface
console	**console**(4S)	console driver and terminal emulator
db	**db**(4M)	SunDials STREAMS module
des	**des**(4S)	DES encryption chip interface
dkio	**dkio**(4S)	generic disk control operations
drum	**drum**(4)	paging device
eeprom	**mem**(4S)	main memory and bus I/O space
fb	**fb**(4S)	driver for Sun console frame buffer
fbio	**fbio**(4S)	frame buffer control operations
fd	**fd**(4S)	Disk driver for Floppy Disk Controllers
filio	**filio**(4)	ioctls that operate directly on files, file descriptors, and sockets
fpa	**fpa**(4S)	Sun-3 floating-point accelerator
gpone	**gpone**(4S)	graphics processor
icmp	**icmp**(4P)	Internet Control Message Protocol
ie	**ie**(4S)	Intel 10 Mb/s Ethernet interface
if	**if**(4N)	general properties of network interfaces
inet	**inet**(4F)	Internet protocol family
ip	**ip**(4P)	Internet Protocol
kb	**kb**(4M)	Sun keyboard STREAMS module
kbd	**kbd**(4S)	Sun keyboard
kmem	**mem**(4S)	main memory and bus I/O space
ldterm	**ldterm**(4M)	standard terminal STREAMS module
le	**le**(4S)	LANCE 10Mb/s Ethernet interface
lo	**lo**(4N)	software loopback network interface
lofs	**lofs**(4S)	loopback virtual file system
mcp	**mcp**(4S)	MCP Multiprotocol Communications Processor
mem	**mem**(4S)	main memory and bus I/O space
mouse	**mouse**(4S)	Sun mouse
ms	**ms**(4M)	Sun mouse STREAMS module
mti	**mti**(4S)	Systech MTI-800/1600 multi-terminal interface
mtio	**mtio**(4)	general magnetic tape interface

NFS	**nfs**(4P)	network file system
nit	**nit**(4P)	Network Interface Tap
nit_buf	**nit_buf**(4M)	STREAMS NIT buffering module
nit_if	**nit_if**(4M)	STREAMS NIT device interface module
nif_pf	**nit_pf**(4M)	STREAMS NIT packet filtering module
null	**null**(4)	data sink
openprom	**openprom**(4S)	PROM monitor configuration interface
pp	**pp**(4)	Centronics-compatible parallel printer port
pty	**pty**(4)	pseudo-terminal driver
rfs	**rfs**(4)	remote file sharing service
root	**root**(4S)	pseudo-driver for Sun386i root disk
routing	**routing**(4N)	system supporting for local network packet routing
sbus	**mem**(4S)	main memory and bus I/O space
sd	**sd**(4S)	driver for SCSI disk devices
sockio	**sockio**(4)	ioctls that operate directly on sockets
sr	**sr**(4S)	driver for CDROM SCSI controller
st	**st**(4S)	driver for SCSI tape devices
streamio	**streamio**(4)	STREAMS ioctl commands
taac	**taac**(4S)	Sun applications accelerator
tcp	**tcp**(4P)	Internet Transmission Control Protocol
tcptli	**tcptli**(4P)	TLI-Conforming TCP Stream-Head
termio	**termio**(4)	general terminal interface
tfs	**tfs**(4S)	translucent file service
tm	**tm**(4S)	Tapemaster 1/2 inch tape controller
tmpfs	**tmpfs**(4S)	memory based filesystem
ttcompat	**ttcompat**(4M)	V7 and 4BSD STREAMS compatibility module
tty	**tty**(4)	controlling terminal interface
udp	**udp**(4P)	Internet User Datagram Protocol
unix	**unix**(4F)	UNIX domain protocol family
vd	**vd**(4)	loadable modules interface
vme16d16	**mem**(4S)	main memory and bus I/O space
vme16d32	**mem**(4S)	main memory and bus I/O space
vme24d16	**mem**(4S)	main memory and bus I/O space
vme24d32	**mem**(4S)	main memory and bus I/O space
vme32d16	**mem**(4S)	main memory and bus I/O space
vme32d32	**mem**(4S)	main memory and bus I/O space
vpc	**vpc**(4S)	Systech VPC-2200 Versatec printer/plotter
win	**win**(4S)	Sun window system
xd	**xd**(4S)	Disk driver for Xylogics 7053 SMD Disk Controller
xt	**xt**(4S)	Xylogics 472 1/2 inch tape controller
xy	**xy**(4S)	Disk driver for Xylogics 450 and 451 SMD Disk Controllers
zero	**mem**(4S)	main memory and bus I/O space
zero	**zero**(4S)	source of zeroes
zs	**zs**(4S)	Zilog 8530 SCC serial communications driver

NAME

ar – Archive 1/4 inch Streaming Tape Drive

AVAILABILITY

Sun-3 and Sun-4 systems only.

DESCRIPTION

The Archive tape controller is a Sun 'QIC-II' interface to an Archive streaming tape drive. It provides a standard tape interface to the device, see **mtio**(4), with some deficiencies listed under BUGS below.

The maximum blocksize for the raw device is limited only by available memory.

FILES

/dev/rar*
/dev/nrar* non-rewinding

SEE ALSO

mtio(4)

DIAGNOSTICS

ar*: would not initialize

ar*: already open
The tape can be opened by only one process at a time

ar*: no such drive

ar*: no cartridge in drive

ar*: cartridge is write protected

ar: interrupt from unitialized controller %x

ar*: many retries, consider retiring this

ar*: %b error at block #

ar*: %b error at block #

ar: giving up on Rdy, try

BUGS

The tape cannot reverse direction so the BSF and BSR ioctls are not supported.

The FSR ioctl is not supported.

The system will hang if the tape is removed while running.

When using the raw device, the number of bytes in any given transfer must be a multiple of 512 bytes. If it is not, the device driver returns an error.

The driver will only write an EOF mark on close if the last operation was a write, without regard for the mode used when opening the file. This delete empty files on a raw tape copy operation.

NAME

arp – Address Resolution Protocol

CONFIG

pseudo-device ether

SYNOPSIS

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

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

DESCRIPTION

ARP is a protocol used to dynamically map between Internet Protocol (IP) and 10Mb/s Ethernet addresses. It is used by all the 10Mb/s Ethernet interface drivers. It is not specific to the Internet Protocol or to the 10Mb/s Ethernet, but this implementation currently supports only that combination.

ARP caches IP-to-Ethernet address mappings. When an interface requests a mapping for an address not in the cache, ARP queues the message which requires the mapping and broadcasts a message on the associated network requesting the address mapping. If a response is provided, the new mapping is cached and any pending message is transmitted. ARP will queue 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 which do not use ARP, **ioctl()** requests are provided to enter and delete entries in the IP-to-Ethernet tables.

USAGE

```
#include <sys/sockio.h>
#include <sys/socket.h>
#include <net/if.h>
#include <net/if_arp.h>
struct arpreq arpreq;
ioctl(s, SIOCSARP, (caddr_t)&arpreq);
ioctl(s, SIOCGARP, (caddr_t)&arpreq);
ioctl(s, SIOCDARP, (caddr_t)&arpreq);
```

Each **ioctl()** 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()** requests may be applied to any socket descriptor *s*, but only by the super-user. The **arpreq** structure contains:

```
/*
 * ARP ioctl request
 */
struct arpreq {
        struct sockaddr  arp_pa;        /* protocol address */
        struct sockaddr  arp_ha;        /* hardware address */
        int      arp_flags;             /* flags */
};
/*  arp_flags field values */
#define ATF_COM                 0x2     /* completed entry (arp_ha valid) */
#define  ATF_PERM               0x4     /* permanent entry */
#define  ATF_PUBL               0x8     /* publish (respond for other host) */
#define  ATF_USETRAILERS                0x10    /* send trailer packets to host */
```

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 which may be written are ATF_PERM, ATF_PUBL and ATF_USETRAILERS. ATF_PERM makes the entry permanent if the **ioctl()** call succeeds. The peculiar nature of the ARP tables may cause the **ioctl()** to fail if more than 6 (permanent) IP addresses hash to the same slot. 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 is also used to negotiate the use of trailer IP encapsulations; trailers are an alternate encapsulation used to allow efficient packet alignment for large packets despite variable-sized headers. Hosts which wish to receive trailer encapsulations so indicate by sending gratuitous ARP translation replies along with replies to IP requests; they are also sent in reply to IP translation replies. The negotiation is thus fully symmetrical, in that either or both hosts may request trailers. The **ATF_USETRAILERS** flag is used to record the receipt of such a reply, and enables the transmission of trailer packets to that host.

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

SEE ALSO

ec(4S), **ie**(4S), **inet**(4F), **arp**(8C), **ifconfig**(8C)

Plummer, Dave, "*An Ethernet Address Resolution Protocol -or- Converting Network Protocol Addresses to 48.bit Ethernet Addresses for Transmission on Ethernet Hardware*," RFC 826, Network Information Center, SRI International, Menlo Park, Calif., November 1982. (Sun 800-1059-10)

Leffler, Sam, and Michael Karels, "*Trailer Encapsulations*," RFC 893, Network Information Center, SRI International, Menlo Park, Calif., April 1984.

DIAGNOSTICS

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

ARP has discovered another host on the local network which responds to mapping requests for its own Internet address.

BUGS

ARP packets on the Ethernet use only 42 bytes of data, however, the smallest legal Ethernet packet is 60 bytes (not including CRC). Some systems may not enforce the minimum packet size, others will.

NAME

audio – telephone quality audio device

CONFIG

device-driver audio

AVAILABILITY

This device is available with SPARCstation 1 systems only.

DESCRIPTION

The **audio** device plays and records a single channel of sound using the AM79C30A Digital Subscriber Controller chip. The chip has a built-in analog to digital converter (ADC) and digital to analog converter (DAC) that can drive either the built-in speaker or an external headphone jack, selectable under software control. Digital audio data is sampled at a rate of 8000 samples per second with 12-bit precision, though the data is compressed, using *u*–law encoding, to 8-bit samples. The resulting audio data quality is equivalent to that of standard telephone service.

The **audio** driver is implemented as a STREAMS device. In order to record audio input, applications **open**(2V) the **/dev/audio** device and read data from it using the **read**(2V) system call. Similarly, sound data is queued to the audio output port by using the **write**(2V) system call.

Opening the Audio Device

The audio device is treated as an exclusive resource: only one process may typically open the device at a time. However, two processes may simultaneously access the device if one opens it read-only and the other opens it write-only.

When a process cannot open **/dev/audio** because the requested access mode is busy:

- if the **O_NDELAY** flag is set in the **open()** *flags* argument, then **open()** returns –1 immediately, with *errno* set to EBUSY.
- if **O_NDELAY** is not set, then **open()** hangs until the device is available or a signal is delivered to the process, in which case **open()** returns –1 with *errno* set to EINTR.

Since the audio device grants exclusive read or write access to a single process at a time, long-lived audio applications may choose to close the device when they enter an idle state, reopening it when required. The *play.waiting* and *record.waiting* flags in the audio information structure (see below) provide an indication that another process has requested access to the device. This information is advisory only; background audio output processes, for example, may choose to relinquish the audio device whenever another process requests write access.

Recording Audio Data

The **read()** system call copies data from the system buffers to the application. Ordinarily, **read()** blocks until the user buffer is filled. The **FIONREAD ioctl** (see **filio**(4)) may be used to determine the amount of data that may be read without blocking. The device may alternatively be set to a non-blocking mode, in which case **read()** completes immediately, but may return fewer bytes than requested. Refer to the **read**(2V) manual page for a complete description of this behavior.

When the audio device is opened with read access, the device driver immediately starts buffering audio input data. Since this consumes system resources, processes that do not record audio data should open the device write-only (**O_WRONLY**).

The transfer of input data to STREAMS buffers may be paused (or resumed) by using the **AUDIO_SETINFO ioctl** to set (or clear) the *record.pause* flag in the audio information structure (see below). All unread input data in the STREAMS queue may be discarded by using the **I_FLUSH** STREAMS **ioctl** (see **streamio**(4)).

Input data accumulates in STREAMS buffers at a rate of 8000 bytes per second. If the application that consumes the data cannot keep up with this data rate, the STREAMS queue may become full. When this occurs, the *record.error* flag is set in the audio information structure and input sampling ceases until there is room in the input queue for additional data. In such cases, the input data stream contains a discontinuity. For this reason, audio recording applications should open the audio device when they are prepared to begin reading data, rather than at the start of extensive initialization.

Playing Audio Data

The **write( )** system call copies data from an applications buffer to the STREAMS output queue. Ordinarily, **write( )** blocks until the entire user buffer is transferred. The device may alternatively be set to a non-blocking mode, in which case **write( )** completes immediately, but may have transferred fewer bytes than requested (see **write**(2V)).

Although **write( )** returns when the data is successfully queued, the actual completion of audio output may take considerably longer. The **AUDIO_DRAIN ioctl** may be issued to allow an application to block until all of the queued output data has been played. Alternatively, a process may request asynchronous notification of output completion by writing a zero-length buffer (end-of-file record) to the output stream. When such a buffer has been processed, the *play.eof* flag in the audio information structure (see below) is incremented.

The final **close( )** of the file descriptor hangs until audio output has drained. If a signal interrupts the **close( )** , or if the process exits without closing the device, any remaining data queued for audio output is flushed and the device is closed immediately.

The conversion of output data may be paused (or resumed) by using the **AUDIO_SETINFO ioctl** to set (or clear) the *play.pause* flag in the audio information structure. Queued output data may be discarded by using the **I_FLUSH** STREAMS **ioctl**.

Output data is played from the STREAMS buffers at a rate of 8000 bytes per second. If the output queue becomes empty, the *play.error* flag is set in the audio information structure and output ceases until additional data is written.

Asynchronous I/O

The **I_SETSIG** STREAMS **ioctl** may be used to enable asynchronous notification, via the **SIGPOLL** signal, of input and output ready conditions. This, in conjunction with non-blocking **read( )** and **write( )** requests, is normally sufficient for applications to maintain an audio stream in the background. Alternatively, asynchronous reads and writes may be initiated using the **aioread**(3) functions.

Audio Data Encoding

The data samples processed by the audio device are encoded in 8 bits. The high-order bit is a sign bit: 1 represents positive data and 0 represents negative data. The low-order 7 bits represent signal magnitude and are inverted (1's complement). The magnitude is encoded according to a μ-law transfer function; such an encoding provides an improved signal-to-noise ratio at low amplitude levels. In order to achieve best results, the audio recording gain should be set so that typical amplitude levels lie within approximately three-fourths of the full dynamic range.

Audio Control Pseudo-Device

It is sometimes convenient to have an application, such as a volume control panel, modify certain characteristics of the audio device while it is being used by an unrelated process. The **/dev/audioctl** minor device is provided for this purpose. Any number of processes may open **/dev/audioctl** simultaneously. However, **read( )** and **write( )** system calls are ignored by **/dev/audioctl**. The **AUDIO_GETINFO** and **AUDIO_SETINFO ioctl** commands may be issued to **/dev/audioctl** in order to determine the status or alter the behavior of **/dev/audio**.

Audio Status Change Notification

Applications that open the audio control pseudo-device may request asynchronous notification of changes in the state of the audio device by setting the **S_MSG** flag in an **I_SETSIG** STREAMS **ioctl**. Such processes receive a **SIGPOLL** signal when any of the following events occurs:

- An **AUDIO_SETINFO ioctl** has altered the device state.
- An input overflow or output underflow has occurred.
- An end-of-file record (zero-length buffer) has been processed on output.
- An **open( )** or **close( )** of **/dev/audio** has altered the device state.

Audio Information Structure

The state of the audio device may be polled or modified using the **AUDIO_GETINFO** and **AUDIO_SETINFO ioctl** commands. These commands operate on the **audio_info** structure, defined in **<sun/audioio.h>** as follows:

```
/* Data encoding values, used below in the encoding field */
#define AUDIO_ENCODING_ULAW     (1)     /* u-law encoding */
#define AUDIO_ENCODING_ALAW     (2)     /* A-law encoding */

/* These ranges apply to record, play, and monitor gain values */
#define AUDIO_MIN_GAIN          (0)             /* minimum gain value */
#define AUDIO_MAX_GAIN          (255)           /* maximum gain value */

/* Audio I/O channel status, used below in the audio_info structure */
struct audio_prinfo {
        /* The following values describe the audio data encoding */
        unsigned                sample_rate;    /* samples per second */
        unsigned                channels;       /* number of interleaved channels */
        unsigned                precision;      /* number of bits per sample */
        unsigned                encoding;       /* data encoding method */

        /* The following values control audio device configuration */
        unsigned                gain;           /* gain level */
        unsigned                port;           /* selected I/O port */

        /* The following values describe the current device state */
        unsigned                samples;        /* number of samples converted */
        unsigned                eof;            /* End Of File counter (play only) */
        unsigned char           pause;          /* non-zero if paused, zero to resume */
        unsigned char           error;          /* non-zero if overflow/underflow */
        unsigned char           waiting;        /* non-zero if a process wants access */

        /* The following values are read-only device state flags */
        unsigned char           open;           /* non-zero if open access granted */
        unsigned char           active;         /* non-zero if I/O active */
};

/* This structure is used in AUDIO_GETINFO and AUDIO_SETINFO ioctl commands */
typedef struct audio_info {
        struct audio_prinfo     record;         /* input status information */
        struct audio_prinfo     play;           /* output status information */
        unsigned                monitor_gain;   /* input to output mix */
} audio_info_t;
```

The *play.gain* and *record.gain* fields specify the output and input volume levels. A value of **AUDIO_MAX_GAIN** indicates maximum gain. The device also allows input data to be monitored by mixing audio input onto the output channel. The *monitor_gain* field controls the level of this feedback path. The *play.port* field controls the output path for the audio device. It may be set to either **AUDIO_SPEAKER** or **AUDIO_HEADPHONE** to direct output to the built-in speaker or the headphone jack, respectively.

The *play.pause* and *record.pause* flags may be used to pause and resume the transfer of data between the audio device and the STREAMS buffers. The *play.error* and *record.error* flags indicate that data underflow or overflow has occurred. The *play.active* and *record.active* flags indicate that data transfer is currently active in the corresponding direction.

The *play.open* and *record.open* flags indicate that the device is currently open with the corresponding access permission. The *play.waiting* and *record.waiting* flags provide an indication that a process may be waiting to access the device. These flags are set automatically when a process blocks on **open()**, though they may also be set using the **AUDIO_SETINFO ioctl** command. They are cleared only when a process relinquishes access by closing the device.

The *play.samples* and *record.samples* fields are initialized, at **open()**, to zero and increment each time a data sample is copied to or from the associated STREAMS queue. Applications that keep track of the number of samples read or written may use these fields to determine exactly how many samples remain in the STREAMS buffers. The *play.eof* field increments whenever a zero-length output buffer is synchronously processed. Applications may use this field to detect the completion of particular segments of audio output.

The *sample_rate*, *channels*, *precision*, and *encoding* fields report the audio data format in use by the device. For now, these values are read-only; however, future audio device implementations may support more than one data encoding format, in which case applications might be able to modify these fields.

Filio and STREAMS IOCTLS

All of the **filio**(4) and **streamio**(4) **ioctl** commands may be issued for the **/dev/audio** device. Because the **/dev/audioctl** device has its own STREAMS queues, most of these commands neither modify nor report the state of **/dev/audio** if issued for the **/dev/audioctl** device. The **I_SETSIG ioctl** may be issued for **/dev/audioctl** to enable the notification of audio status changes, as described above.

Audio IOCTLS

The audio device additionally supports the following **ioctl** commands:

AUDIO_DRAIN

The argument is ignored. This command suspends the calling process until the output STREAMS queue is empty, or until a signal is delivered to the calling process. It may only be issued for the **/dev/audio** device. An implicit **AUDIO_DRAIN** is performed on the final **close()** of **/dev/audio**.

AUDIO_GETINFO

The argument is a pointer to an **audio_info** structure. This command may be issued for either **/dev/audio** or **/dev/audioctl**. The current state of the **/dev/audio** device is returned in the structure.

AUDIO_SETINFO

The argument is a pointer to an **audio_info** structure. This command may be issued for either **/dev/audio** or **/dev/audioctl**. This command configures the audio device according to the structure supplied and overwrites the structure with the new state of the device. [Note: The *play.samples*, *record.samples*, *play.error*, *record.error*, and *play.eof* fields are modified to reflect the state of the device when the **AUDIO_SETINFO** was issued. This allows programs to atomically modify these fields while retrieving the previous value.]

Certain fields in the information structure, such as the *pause* flags, are treated as read-only when **/dev/audio** is not open with the corresponding access permission. Other fields, such as the gain levels and encoding information, may have a restricted set of acceptable values. Applications that attempt to modify such fields should check the returned values to be sure that the corresponding change took effect.

Once set, the following values persist through subsequent **open()** and **close()** calls of the device: *play.gain*, *record.gain*, *monitor_gain*, *play.port*, and *record.port*. All other state is reset when the corresponding I/O stream of **/dev/audio** is closed.

The **audio_info** structure may be initialized through the use of the **AUDIO_INITINFO** macro. This macro sets all fields in the structure to values that are ignored by the **AUDIO_SETINFO** command. For instance, the following code switches the output port from the built-in speaker to the headphone jack without modifying any other audio parameters:

```
audio_info_t    info;

AUDIO_INITINFO(&info);
info.play.port = AUDIO_HEADPHONE;
err = ioctl(audio_fd, AUDIO_SETINFO, &info);
```

This technique is preferred over using a sequence of **AUDIO_GETINFO** followed by **AUDIO_SETINFO**.

Unsupported Device Control Features

The AM79C30A chip is capable of a performing a number of functions that are not currently supported by the device driver, many of which were designed primarily for telephony applications. For example, the chip can generate ringer tones and has a number of specialized filtering capabilities that are designed to compensate for different types of external speakers and microphones.

Ordinarily, applications do not need to access these capabilities and, further, altering the chip's characteristics may interfere with its normal behavior. However, knowledgeable applications may use the unsupported **AUDIOGETREG** and **AUDIOSETREG** **ioctl** commands to read and write the chip registers directly. The description of this interface may be found in **<sbusdev/audio_79C30.h>**. Note: these commands are supplied for prototyping purposes only and may become obsolete in a future release of the audio driver.

FILES

/dev/audio
/dev/audioctl
/usr/demo/SOUND

SEE ALSO

ioctl(2), **poll**(2), **read**(2V), **write**(2V), **aioread**(3), **filio**(4), **streamio**(4)

AMD data sheet for the AM79C30A Digital Subscriber Controller, Publication number 09893.

BUGS

Due to a *feature* of the STREAMS implementation, programs that are terminated or exit without closing the **audio** device may hang for a short period while audio output drains. In general, programs that produce audio output should catch the **SIGINT** signal and flush the output stream before exiting.

The current driver implementation does not support the A-law encoding mode of the AM79C30A chip. Future implementations may permit the **AUDIO_SETINFO** **ioctl** to modify the *play.encoding* and *record.encoding* fields of the device information structure to enable this mode.

FUTURE DIRECTIONS

Workstation audio resources should be managed by a networked audio server, in the same way that the video monitor is manipulated by a window system server. For the time being, we encourage you to write your programs in a modular fashion, isolating the **audio** device-specific functions, so that they may be easily ported to such an environment.

NAME

bwtwo – black and white memory frame buffer

CONFIG — SUN-3, SUN-3x SYSTEMS

device bwtwo0 at obmem 1 csr 0xff000000 priority 4
device bwtwo0 at obmem 2 csr 0x100000 priority 4
device bwtwo0 at obmem 3 csr 0xff000000 priority 4
device bwtwo0 at obmem 4 csr 0xff000000
device bwtwo0 at obmem 7 csr 0xff000000 priority 4
device bwtwo0 at obmem ? csr 0x50300000 priority 4

The first synopsis line given above is used to generate a kernel for Sun-3/75, Sun-3/140 or Sun-3/160 systems; the second, for a Sun-3/50 system; the third, for a Sun-3/260 system; the fourth, for a Sun-3/110 system; the fifth, for a Sun-3/60 system; and the sixth for Sun-3/80 and Sun-3/470 systems.

CONFIG — SUN-4 SYSTEMS

device bwtwo0 at obio 1 csr 0xfd000000 priority 4
device bwtwo0 at obio 2 csr 0xfb300000 priority 4
device bwtwo0 at obio 3 csr 0xfb300000 priority 4
device bwtwo0 at obio 4 csr 0xfb300000 priority 4

The first synopsis line given above should be used to generate a kernel for a Sun-4/260 or Sun-4/280 system; the second, for a Sun-4/110 system; the third for a Sun-4/330 system; and the fourth for a Sun-4/460 system.

CONFIG — SPARCstation 1 SYSTEMS

device-driver bwtwo

CONFIG — Sun386i SYSTEM

device bwtwo0 at obmem ? csr 0xA0200000

DESCRIPTION

The **bwtwo** interface provides access to Sun monochrome memory frame buffers. It supports the ioctls described in **fbio**(4S).

If **flags 0x1** is specified, frame buffer write operations are buffered through regular high-speed RAM. This "copy memory" mode of operation speeds frame buffer accesses, but consumes an extra 128K bytes of memory. Only Sun-3/75, Sun-3/140, and Sun-3/160 systems support copy memory; on other systems a warning message is printed and the flag is ignored.

Reading or writing to the frame buffer is not allowed — you must use the **mmap**(2) system call to map the board into your address space.

FILES

/dev/bwtwo[0-9] device files

SEE ALSO

mmap(2), **cgfour**(4S), **fb**(4S), **fbio**(4S)

BUGS

Use of vertical-retrace interrupts is not supported.

NAME

cdromio – CDROM control operations

DESCRIPTION

The Sun CDROM device driver supports a set of **ioctl**(2) commands for audio operations and CDROM specific operations. It also supports the **dkio**(4S) operations — generic disk control operation for all Sun disk drivers. See **dkio**(4S) Basic to these **cdromio ioctl**() requests are the definitions in **<scsi/targets/srdef.h>** or **<sundev/srreg.h>**

```
/*
 * CDROM I/O controls type definitions
 */

/* definition of play audio msf structure */
struct cdrom_msf {
        unsigned char   cdmsf_min0;     /* starting minute */
        unsigned char   cdmsf_sec0;     /* starting second */
        unsigned char   cdmsf_frame0;   /* starting frame  */
        unsigned char   cdmsf_min1;     /* ending minute   */
        unsigned char   cdmsf_sec1;     /* ending second   */
        unsigned char   cdmsf_frame1;   /* ending frame    */
};

/* definition of play audio track/index structure */
struct cdrom_ti {
        unsigned char   cdti_trk0;      /* starting track */
        unsigned char   cdti_ind0;      /* starting index */
        unsigned char   cdti_trk1;      /* ending track */
        unsigned char   cdti_ind1;      /* ending index */
};

/* definition of read toc header structure */
struct cdrom_tochdr {
        unsigned char   cdth_trk0;      /* starting track */
        unsigned char   cdth_trk1;      /* ending track */
};

/* definition of read toc entry structure */
struct cdrom_tocentry {
        unsigned char   cdte_track;
        unsigned char   cdte_adr        :4;
        unsigned char   cdte_ctrl       :4;
        unsigned char   cdte_format;
        union {
                struct {
                        unsigned char   minute;
                        unsigned char   second;
                        unsigned char   frame;
                } msf;
                int     lba;
        } cdte_addr;
        unsigned char   cdte_datamode;
};
```

```
/*
 * Bitmask for CDROM data track in the cdte_ctrl field
 * A track is either data or audio.
 */
#define CDROM_DATA_TRACK   0x04

/*
 * CDROM address format definition, for use with struct cdrom_tocentry
 */
#define CDROM_LBA      0x01
#define CDROM_MSF      0x02

/*
 * For CDROMREADTOCENTRY, set the cdte_track to CDROM_LEADOUT to get
 * the information for the leadout track.
 */
#define CDROM_LEADOUT        0xAA

struct cdrom_subchnl {
        unsigned char   cdsc_format;
        unsigned char   cdsc_audiostatus;
        unsigned char   cdsc_adr:       4;
        unsigned char   cdsc_ctrl:      4;
        unsigned char   cdsc_trk;
        unsigned char   cdsc_ind;
        union {
                struct {
                        unsigned char   minute;
                        unsigned char   second;
                        unsigned char   frame;
                } msf;
                int     lba;
        } cdsc_absaddr;
        union {
                struct {
                        unsigned char   minute;
                        unsigned char   second;
                        unsigned char   frame;
                } msf;
                int     lba;
        } cdsc_reladdr;
};

/*
 * Definition for audio status returned from Read Sub-channel
 */
#define CDROM_AUDIO_INVALID    0x00   /* audio status not supported */
#define CDROM_AUDIO_PLAY       0x11   /* audio play operation in progress */
#define CDROM_AUDIO_PAUSED     0x12   /* audio play operation paused */
#define CDROM_AUDIO_COMPLETED  0x13   /* audio play successfully completed */
#define CDROM_AUDIO_ERROR      0x14   /* audio play stopped due to error */
#define CDROM_AUDIO_NO_STATUS  0x15   /* no current audio status to return */
```

```
/* definition of audio volume control structure */
struct cdrom_volctrl {
        unsigned char   cdvc_chnl0;
        unsigned char   cdvc_chnl1;
        unsigned char   cdvc_chnl2;
        unsigned char   cdvc_chnl3;
};

struct cdrom_read {
    int    cdread_lba;
    caddr_t cdread_bufaddr;
    int    cdread_buflen;
};

#define CDROM_MODE1_SIZE      2048
#define CDROM_MODE2_SIZE      2336

/*
 * CDROM I/O control commands
 */
#define CDROMPAUSE  _IO(c, 10)  /* Pause Audio Operation */

#define CDROMRESUME  _IO(c, 11)          /* Resume paused Audio Operation */

#define CDROMPLAYMSF _IOW(c, 12, struct cdrom_msf)    /* Play Audio MSF */

#define CDROMPLAYTRKIND  _IOW(c, 13, struct cdrom_ti)  /* Play Audio Trk/ind */

#define CDROMREADTOCHDR  _IOR(c, 103, struct cdrom_tochdr) /* Read TOC hdr */

#define CDROMREADTOCENTRY _IOWR(c, 104, struct cdrom_tocentry) /* Read TOC */

#define CDROMSTOP       _IO(c, 105)          /* Stop the cdrom drive */

#define CDROMSTART      _IO(c, 106)          /* Start the cdrom drive */

#define CDROMEJECT      _IO(c, 107)          /* Ejects the cdrom caddy */

#define CDROMVOLCTRL    _IOW(c, 14, struct cdrom_volctrl) /* volume control */

#define CDROMSUBCHNL  _IOWR(c, 108, struct cdrom_subchnl) /* read subchannel */

#define CDROMREADMODE2        _IOW(c, 110, struct cdrom_read) /* mode 2 */

#define CDROMREADMODE1        _IOW(c, 111, struct cdrom_read) /* mode 1 */
```

The **CDROMPAUSE** **ioctl()** pauses the current audio play operation and the **CDROMRESUME** **ioctl()** resumes the paused audio play operation. The **CDROMSTART** **ioctl()** spins up the disc and seeks to the last address requested, while the **CDROMSTOP** **ioctl()** spins down the disc and the **CDROMEJECT** **ioctl()** ejects the caddy with the disc. All of the above **ioctl()** calls only take a file descriptor and a command as arguments. They have the form:

```
ioctl(fd, cmd)
        int     fd;
        int     cmd;
```

The rest of the **ioctl( )** calls have the form:

```
ioctl(fd, cmd, ptr)
        int     fd;
        int     cmd;
        char    *ptr;
```

where *ptr* is a pointer to a struct or an integer.

The **CDROMPLAYMSF ioctl( )** command requests the drive to output the audio signals staring at the specified starting address and continue the audio play until the specified ending address is detected. The address is in **MSF** (minute, second, frame) format. The third argument of the function call is a pointer to the type **struct cdrom_msf**.

The **CDROMPLAYTRKIND ioctl( )** command is similar to **CDROMPLAYMSF**. The starting and ending address is in track/index format. The third argument of the function call is a pointer to the type **struct cdrom_ti**.

The **CDROMREADTOCHDR ioctl( )** command returns the header of the **TOC** (table of contents). The header consists of the starting tracking number and the ending track number of the disc. These two numbers are returned through a pointer of **struct cdrom_tochdr**. While the disc can start at any number, all tracks between the first and last tracks are in contiguous ascending order. A related **ioctl( )** command is **CDROMREADTOCENTRY**. This command returns the information of a specified track. The third argument of the function call is a pointer to the type **struct cdrom_tocentry**. The caller need to supply the track number and the address format. This command will return a 4-bit **adr** field, a 4-bit **ctrl** field, the starting address in **MSF** format or **LBA** format, and the data mode if the track is a data track. The **ctrl** field specifies whether the track is data or audio. To get information for the lead-out area, supply the **ioctl( )** command with the track field set to **CDROM_LEADOUT** (0xAA).

The **CDROMVOLCTRL ioctl( )** command controls the audio output level. The **SCSI** command allows the control of up to 4 channels. The current implementation of the supported **CDROM** drive only uses channel 0 and channel 1. The valid values of volume control are between 0x00 and 0xFF, with a value of 0xFF indicating maximum volume. The third argument of the function call is a pointer to **struct cdrom_volctrl** which contains the output volume values.

The **CDROMSUBCHNL ioctl( )** command reads the Q sub-channel data of the current block. The sub-channel data includes track number, index number, absolute **CDROM** address, track relative **CDROM** address, control data and audio status. All information is returned through a pointer to **struct cdrom_subchnl**. The caller needs to supply the address format for the returned address.

The **CDROMREADMODE2** and **CDROMREADMODE1 ioctl( )** commands are only available on SPARCstation 1 systems.

Finally, on SPARCstation 1 systems only, the driver supports the user **SCSI** command interface. By issuing the **ioctl( )** command, **USCSICMD**, The caller can supply any **SCSI-2** commands that the **CDROM** drive supports. The caller has to provide all the parameters in the **SCSI** command block, as well as other information such as the user buffer address and buffer length. See the definitions in **<scsi/impl/uscsi.h>**. The **ioctl( )** call has the form:

```
ioctl(fd, cmd, ptr)
        int     fd;
        int     cmd;
        char    *ptr;
```

where *ptr* is a pointer to the type:

```
struct uscsi_scmd {
        caddr_t uscsi_cdb;
        int     uscsi_cdblen;
        caddr_t uscsi_bufaddr;
        int     uscsi_buflen;
        unsigned char   uscsi_status;
        int     uscsi_flags;
};
```

uscsi_cdb is a pointer to the SCSI command block. Group 0 **cdb's** are 6 bytes long while the other groups are 10 bytes or 12 bytes. **uscsi_cdblen** is the length of the **cdb**. **uscsi_bufaddr** is the pointer to the user buffer for parameter passing or data input/output. *buflen* is the length of the user buffer. **uscsi_flags** are the execution flags for SCSI input/output. The possible flags are USCSI_SILENT, USCSI_DIAGNOSE, USCSI_ISOLATE, USCSI_READ, and USCSI_WRITE.

FILES

/usr/include/scsi/targets/srdef.h
/usr/include/scsi/impl/uscsi.h
/usr/include/sundev/srreg.h

SEE ALSO

ioctl(2), **dkio**(4S), sr(4S)

BUGS

The interface to this device is preliminary and subject to change in future releases. You are encouraged to write your programs in a modular fashion so that you can easily incorporate future changes.

NAME

cgeight – 24-bit color memory frame buffer

CONFIG — SUN-3 AND SUN-4 SYSTEMS

device cgeight0 at obmem 7 csr 0xff300000 priority 4
device cgeight0 at obio 4 csr 0xfb300000 priority 4

The first synopsis line should be used to generate a kernel for the Sun-3/60; the second synopsis for a Sun-4/110 or Sun-4/150 system.

CONFIG — SUN-3x SYSTEM

device cgeight0 at obio ? csr 0x50300000 priority 4

DESCRIPTION

The **cgeight** is a 24-bit color memory frame buffer with a monochrome overlay plane and an overlay enable plane implemented optionally on the Sun-4/110, Sun-4/150, Sun-3/60, Sun-3/470 and Sun-3/80 system models. It provides the standard frame buffer interface as defined in **fbio**(4S).

In addition to the ioctls described under **fbio**(4S), the **cgeight** interface responds to two **cgeight**-specific colormap ioctls, **FBIOPUTCMAP** and **FBIOGETCMAP**. **FBIOPUTCMAP** returns no information other than success/failure using the ioctl return value. **FBIOGETCMAP** returns its information in the arrays pointed to by the red, green, and blue members of its **fbcmap** structure argument; **fbcmap** is defined in **<sun/fbio.h>** as:

```
struct fbcmap {
        int             index;          /* first element (0 origin) */
        int             count;          /* number of elements */
        unsigned char   *red;           /* red color map elements */
        unsigned char   *green;         /* green color map elements */
        unsigned char   *blue;          /* blue color map elements */
};
```

The driver uses color board vertical-retrace interrupts to load the colormap.

The systems have an overlay plane colormap, which is accessed by encoding the plane group into the index value with the **PIX_GROUP** macro (see **<pixrect/pr_planegroups.h>**).

When using the **mmap** system call to map in the **cgeight** frame buffer. The device looks like:

```
DACBASE: 0x200000    -> Brooktree Ramdac          16 bytes
         0x202000    -> P4 Regiter                4 bytes
OVLBASE: 0x210000    -> Overlay Plane             1152x900x1
         0x230000    -> Overlay Enable Planea     1152x900x1
         0x250000    -> 24-bit Frame Buffera      1152x900x32
```

FILES

/dev/cgeight0
<sun/fbio.h>
<pixrect/pr_planegroups.h>

SEE ALSO

mmap(2), **fbio**(4S)

NAME

cgfour – Sun-3 color memory frame buffer

CONFIG — SUN-3 SYSTEMS

device cgfour0 at obmem 4 csr 0xff000000 priority 4
device cgfour0 at obmem 7 csr 0xff300000 priority 4

The first synopsis line given should be used to generate a kernel for the Sun-3/110 system; and the second, for a Sun-3/60 system.

CONFIG — SUN-3x SYSTEMS

device cgfour0 at obmem ? csr 0x50300000 priority 4

CONFIG — SUN-4 SYSTEMS

device cgfour0 at obio 2 csr 0xfb300000 priority 4
device cgfour0 at obio 3 csr 0xfb300000 priority 4
device cgfour0 at obio 4 csr 0xfb300000 priority 4

The first synopsis line given should be used to generate a kernel for the Sun-4/110 system; the second, for a Sun-4/330 system; and the third for a Sun-4/460 system.

DESCRIPTION

The **cgfour** is a color memory frame buffer with a monochrome overlay plane and an overlay enable plane implemented on the Sun-3/110 system and some Sun-3/60 system models. It provides the standard frame buffer interface as defined in **fbio**(4S).

In addition to the ioctls described under **fbio**(4S), the **cgfour** interface responds to two **cgfour**-specific colormap ioctls, **FBIOPUTCMAP** and **FBIOGETCMAP**. **FBIOPUTCMAP** returns no information other than success/failure using the ioctl return value. **FBIOGETCMAP** returns its information in the arrays pointed to by the red, green, and blue members of its **fbcmap** structure argument; **fbcmap** is defined in **<sun/fbio.h>** as:

```
struct fbcmap {
        int             index;          /* first element (0 origin) */
        int             count;          /* number of elements */
        unsigned char   *red;           /* red color map elements */
        unsigned char   *green;         /* green color map elements */
        unsigned char   *blue;          /* blue color map elements */
};
```

The driver uses color board vertical-retrace interrupts to load the colormap.

The Sun-3/60 system has an overlay plane colormap, which is accessed by encoding the plane group into the index value with the **PIX_GROUP** macro (see **<pixrect/pr_planegroups.h>**).

FILES

/dev/cgfour0

SEE ALSO

mmap(2), **fbio**(4S)

NAME

cgnine – 24-bit VME color memory frame buffer

CONFIGURATION

device cgnine0 at vme32d32 ? csr 0x08000000 priority 4 vector cgnineintr 0xaa

DESCRIPTION

cgnine is a 24-bit double-buffered VME-based color frame buffer. It provides the standard frame buffer interface defined in **fbio**(4S), and and can be paired with the **GP2** graphics accelerator board using **gpconfig**(8).

cgnine has two bits of overlay planes, each of which is a 1-bit deep frame buffer that overlays the 24-bit plane group. When either bit of the two overlay planes is non-zero, the pixel shows the color of the overlay plane. If both bits are zero, the color frame buffer underneath is visible.

The 24-bit frame buffer pixel is organized as one longword (32 bits) per pixel. The pixel format is defined in **<pixrect/pixrect.h>** as follows:

```
union fbunit {
unsigned int    packed; /* whole-sale deal */
struct {
        unsigned int    A:8;    /* unused, for now */
        unsigned int    B:8;    /* blue channel */
        unsigned int    G:8;    /* green channel */
        unsigned int    R:8;    /* red channel */
        }
                channel;        /* access per channel */
};
```

When the board is in double-buffer mode, the low 4 bits of each channel are ignored when written to, which yields 12-bit double-buffering.

The higher bit of the overlay planes ranges from offset 0 to 128K (0x20000) bytes. The lower bit ranges from 128K to 256K bytes. The 4MB (0x400000) of the 24-bit deep pixels begins at 256K. The addresses of the control registers start at the next page after the 24-bit deep pixels.

FILES

/dev/cgnine0	device special file
/dev/gpone0a	**cgnine** bound with **GP2**
/dev/fb	default frame buffer

SEE ALSO

mmap(2), **fbio**(4S), **gpone**(4S) **gpconfig**(8)

NAME

cgsix – accelerated 8-bit color frame buffer

CONFIG — SUN-3, SUN-3x, SUN-4 SYSTEMS

device cgsix0 at obmem ? csr 0xff000000 priority 4
device cgsix0 at obmem ? csr 0x50000000 priority 4
device cgsix0 at obio ? csr 0xfb000000 priority 4

The first synopsis line given should be used for Sun-3/60 systems, the second for Sun-3x systems, and the third for Sun-4 systems.

CONFIG — SPARCstation 1 SYSTEMS

device-driver cgsix

DESCRIPTION

The **cgsix** is a low-end graphics accelerator designed to enhance vector and polygon drawing performance. It has an 8-bit color frame buffer and provides the standard frame buffer interface as defined in **fbio**(4S).

The cgsix has registers and memory that may be mapped with **mmap**(2), using the offsets defined in **<sundev/cg6reg.h>**.

FILES

/dev/cgsix0

SEE ALSO

mmap(2), **fbio**(4S)

NAME

cgthree – 8-bit color memory frame buffer

CONFIG — SPARCstation 1 SYSTEMS

device-driver cgthree

CONFIG — Sun386i SYSTEM

device cgthree0 at obmem ? csr 0xA0400000

AVAILABILITY

SPARCstation 1 and Sun386i systems only.

DESCRIPTION

cgthree is a color memory frame buffer. It provides the standard frame buffer interface as defined in **fbio**(4S).

FILES

/dev/cgthree[0-9]

SEE ALSO

mmap(2), **fbio**(4S)

NAME

cgtwo – color graphics interface

CONFIG — SUN-3, SUN-3x, SUN-4 SYSTEMS

cgtwo0 at vme24d16 ? csr 0x400000 priority 4 vector cgtwointr 0xa8

DESCRIPTION

The **cgtwo** interface provides access to the color graphics controller board, which is normally supplied with a 19'' 66 Hz non-interlaced color monitor. It provides the standard frame buffer interface as defined in **fbio**(4S).

The hardware consumes 4 megabytes of VME bus address space. The board starts at standard address 0x400000. The board must be configured for interrupt level 4.

FILES

/dev/cgtwo[0-9]

SEE ALSO

mmap(2), **fbio**(4S)

NAME

clone – open any minor device on a STREAMS driver

DESCRIPTION

clone is a STREAMS software driver that finds and opens an unused minor device on another STREAMS driver. The minor device passed to **clone** during the open operation is interpreted as the major device number of another STREAMS driver for which an unused minor device is to be obtained. Each such open results in a separate stream to a previously unused minor device.

The **clone** driver supports only an **open**(2V) function. This open function performs all of the necessary work so that subsequent system calls (including **close**(2V)) require no further involvement of the **clone** driver.

ERRORS

clone generates an ENXIO error, without opening the device, if the minor device number provided does not correspond to a valid major device, or if the driver indicated is not a STREAMS driver.

WARNINGS

Multiple opens of the same minor device are not supported through the **clone** interface. Executing **stat**(2V) on the file system node for a cloned device yields a different result than does executing **fstat** using a file descriptor obtained from opening that node.

SEE ALSO

close(2V), **open**(2V), **stat**(2V)

NAME

console – console driver and terminal emulator for the Sun workstation

CONFIG

None; included in standard system.

SYNOPSIS

```
#include <fcntl.h>
#include <sys/termios.h>
open("/dev/console", mode);
```

DESCRIPTION

console is an indirect driver for the Sun console terminal. On a Sun workstation, this driver refers to the workstation console driver, which implements a standard UNIX system terminal. On a Sun server without a keyboard or a frame buffer, this driver refers to the CPU serial port driver (**zs**(4S)); a terminal is normally connected to this port.

The workstation console does not support any of the **termio**(4) device control functions specified by flags in the **c_cflag** word of the **termios** structure or by the **IGNBRK**, **IGNPAR**, **PARMRK**, or **INPCK** flags in the **c_iflag** word of the **termios** structure, as these functions apply only to asynchronous serial ports. All other **termio**(4) functions must be performed by STREAMS modules pushed atop the driver; when a slave device is opened, the **ldterm**(4M) and **ttcompat**(4M) STREAMS modules are automatically pushed on top of the stream, providing the standard **termio**(4) interface.

The workstation console driver calls the PROM resident monitor to output data to the console frame buffer. Keystrokes from the CPU serial port to which the keyboard is connected are routed through the keyboard STREAMS module (**kb**(4M)) and treated as input.

When the Sun window system **win**(4S) is active, console input is directed through the window system rather than being treated as input by the workstation console driver.

IOCTLS

An ioctl TIOCCONS can be applied to pseudo-terminals (**pty**(4)) to route output that would normally appear on the console to the pseudo-terminal instead. Thus, the window system does a **TIOCCONS** on a pseudo-terminal so that the system will route console output to the window to which that pseudo-terminal is connected, rather than routing output through the PROM monitor to the screen, since routing output through the PROM monitor destroys the integrity of the screen. Note: when you use **TIOCCONS** in this way, the console *input* is routed from the pseudo-terminal as well.

If a **TIOCCONS** is performed on **/dev/console**, or the pseudo-terminal to which console output is being routed is closed, output to the console will again be routed to the workstation console driver.

ANSI STANDARD TERMINAL EMULATION

The Sun Workstation's PROM monitor provides routines that emulates a standard ANSI X3.64 terminal.

Note: the VT100 also follows the ANSI X3.64 standard but both the Sun and the VT100 have nonstandard extensions to the ANSI X3.64 standard. The Sun terminal emulator and the VT100 are *not* compatible in any true sense.

The Sun console displays 34 lines of 80 ASCII characters per line, with scrolling, (x, y) cursor addressability, and a number of other control functions.

The Sun console displays a non-blinking block cursor which marks the current line and character position on the screen. ASCII characters between 0x20 (space) and 0x7E (tilde) inclusive are printing characters — when one is written to the Sun console (and is not part of an escape sequence), it is displayed at the current cursor position and the cursor moves one position to the right on the current line. If the cursor is already at the right edge of the screen, it moves to the first character position on the next line. If the cursor is already at the right edge of the screen on the bottom line, the Line-feed function is performed (see CTRL-J below), which scrolls the screen up by one or more lines or wraps around, before moving the cursor to the first character position on the next line.

Control Sequence Syntax

The Sun console defines a number of control sequences which may occur in its input. When such a sequence is written to the Sun console, it is not displayed on the screen, but effects some control function as described below, for example, moves the cursor or sets a display mode.

Some of the control sequences consist of a single character. The notation

CTRL-*X*

for some character *X* , represents a control character.

Other ANSI control sequences are of the form

ESC [*params*char

Spaces are included only for readability; these characters must occur in the given sequence without the intervening spaces.

ESC represents the ASCII escape character (ESC, CTRL-[, 0x1B).

[The next character is a left square bracket '[' (0x5B).

params are a sequence of zero or more decimal numbers made up of digits between 0 and 9, separated by semicolons.

char represents a function character, which is different for each control sequence.

Some examples of syntactically valid escape sequences are (again, ESC represent the single ASCII character 'Escape'):

ESC[m	*select graphic rendition with default parameter*
ESC[7m	*select graphic rendition with reverse image*
ESC[33;54H	*set cursor position*
ESC[123;456;0;;3;B	*move cursor down*

Syntactically valid ANSI escape sequences which are not currently interpreted by the Sun console are ignored. Control characters which are not currently interpreted by the Sun console are also ignored.

Each control function requires a specified number of parameters, as noted below. If fewer parameters are supplied, the remaining parameters default to 1, except as noted in the descriptions below.

If more than the required number of parameters is supplied, only the last *n* are used, where *n* is the number required by that particular command character. Also, parameters which are omitted or set to zero are reset to the default value of 1 (except as noted below).

Consider, for example, the command character M which requires one parameter. ESC[;M and ESC[0M and ESC[M and ESC[23;15;32;1M are all equivalent to ESC[1M and provide a parameter value of 1. Note: ESC[;5M (interpreted as 'ESC[5M') is *not* equivalent to ESC[5;M (interpreted as 'ESC[5;1M') which is ultimately interpreted as 'ESC[1M').

In the syntax descriptions below, parameters are represented as '**#**' or '**#1;#2**'.

ANSI Control Functions

The following paragraphs specify the ANSI control functions implemented by the Sun console. Each description gives:

- the control sequence syntax
- the hex equivalent of control characters where applicable
- the control function name and ANSI or Sun abbreviation (if any).
- description of parameters required, if any
- description of the control function
- for functions which set a mode, the initial setting of the mode. The initial settings can be restored with the SUNRESET escape sequence.

Control Character Functions

CTRL-G (0x7) Bell (BEL)
The Sun Workstation Model 100 and 100U is not equipped with an audible bell. It 'rings the bell' by flashing the entire screen. The window system flashes the window.

CTRL-H (0x8) Backspace (BS)
The cursor moves one position to the left on the current line. If it is already at the left edge of the screen, nothing happens.

CTRL-I (0x9) Tab (TAB)
The cursor moves right on the current line to the next tab stop. The tab stops are fixed at every multiple of 8 columns. If the cursor is already at the right edge of the screen, nothing happens; otherwise the cursor moves right a minimum of one and a maximum of eight character positions.

CTRL-J (0xA) Line-feed (LF)
The cursor moves down one line, remaining at the same character position on the line. If the cursor is already at the bottom line, the screen either scrolls up or "wraps around" depending on the setting of an internal variable *S* (initially 1) which can be changed by the ESC[r control sequence. If *S* is greater than zero, the entire screen (including the cursor) is scrolled up by *S* lines before executing the line-feed. The top *S* lines scroll off the screen and are lost. *S* new blank lines scroll onto the bottom of the screen. After scrolling, the line-feed is executed by moving the cursor down one line.

If *S* is zero, 'wrap-around' mode is entered. 'ESC [1 r' exits back to scroll mode. If a line-feed occurs on the bottom line in wrap mode, the cursor goes to the same character position in the top line of the screen. When any line-feed occurs, the line that the cursor moves to is cleared. This means that no scrolling occurs. Wrap-around mode is not implemented in the window system.

The screen scrolls as fast as possible depending on how much data is backed up waiting to be printed. Whenever a scroll must take place and the console is in normal scroll mode ('ESC [1 r'), it scans the rest of the data awaiting printing to see how many line-feeds occur in it. This scan stops when any control character from the set {**VT, FF, SO, SI, DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB, CAN, EM, SUB, ESC, FS, GS, RS, US**} is found. At that point, the screen is scrolled by N lines ($N \geq 1$) and processing continues. The scanned text is still processed normally to fill in the newly created lines. This results in much faster scrolling with scrolling as long as no escape codes or other control characters are intermixed with the text.

See also the discussion of the 'Set scrolling' (**ESC[r**) control function below.

CTRL-K (0xB) Reverse Line-feed
The cursor moves up one line, remaining at the same character position on the line. If the cursor is already at the top line, nothing happens.

CTRL-L (0xC) Form-feed (FF)
The cursor is positioned to the Home position (upper-left corner) and the entire screen is cleared.

CTRL-M (0xD) Return (CR)
The cursor moves to the leftmost character position on the current line.

Escape Sequence Functions

CTRL-[(0x1B) Escape (ESC)
This is the escape character. Escape initiates a multi-character control sequence.

ESC[#@ Insert Character (ICH)
Takes one parameter, # (default 1). Inserts # spaces at the current cursor position. The tail of the current line starting at the current cursor position inclusive is shifted to the right by # character positions to make room for the spaces. The rightmost # character positions shift off the line and are lost. The position of the cursor is unchanged.

ESC[#A Cursor Up (CUU)

Takes one parameter, # (default 1). Moves the cursor up # lines. If the cursor is fewer than # lines from the top of the screen, moves the cursor to the topmost line on the screen. The character position of the cursor on the line is unchanged.

ESC[#B Cursor Down (CUD)

Takes one parameter, # (default 1). Moves the cursor down # lines. If the cursor is fewer than # lines from the bottom of the screen, move the cursor to the last line on the screen. The character position of the cursor on the line is unchanged.

ESC[#C Cursor Forward (CUF)

Takes one parameter, # (default 1). Moves the cursor to the right by # character positions on the current line. If the cursor is fewer than # positions from the right edge of the screen, moves the cursor to the rightmost position on the current line.

ESC[#D Cursor Backward (CUB)

Takes one parameter, # (default 1). Moves the cursor to the left by # character positions on the current line. If the cursor is fewer than # positions from the left edge of the screen, moves the cursor to the leftmost position on the current line.

ESC[#E Cursor Next Line (CNL)

Takes one parameter, # (default 1). Positions the cursor at the leftmost character position on the #-th line below the current line. If the current line is less than # lines from the bottom of the screen, positions the cursor at the leftmost character position on the bottom line.

ESC[#1;#2f Horizontal And Vertical Position (HVP)

or

ESC[#1;#2H Cursor Position (CUP)

Takes two parameters, #1 and #2 (default 1, 1). Moves the cursor to the #2-th character position on the #1-th line. Character positions are numbered from 1 at the left edge of the screen; line positions are numbered from 1 at the top of the screen. Hence, if both parameters are omitted, the default action moves the cursor to the home position (upper left corner). If only one parameter is supplied, the cursor moves to column 1 of the specified line.

ESC[J Erase in Display (ED)

Takes no parameters. Erases from the current cursor position inclusive to the end of the screen. In other words, erases from the current cursor position inclusive to the end of the current line and all lines below the current line. The cursor position is unchanged.

ESC[K Erase in Line (EL)

Takes no parameters. Erases from the current cursor position inclusive to the end of the current line. The cursor position is unchanged.

ESC[#L Insert Line (IL)

Takes one parameter, # (default 1). Makes room for # new lines starting at the current line by scrolling down by # lines the portion of the screen from the current line inclusive to the bottom. The # new lines at the cursor are filled with spaces; the bottom # lines shift off the bottom of the screen and are lost. The position of the cursor on the screen is unchanged.

ESC[#M Delete Line (DL)

Takes one parameter, # (default 1). Deletes # lines beginning with the current line. The portion of the screen from the current line inclusive to the bottom is scrolled upward by # lines. The # new lines scrolling onto the bottom of the screen are filled with spaces; the # old lines beginning at the cursor line are deleted. The position of the cursor on the screen is unchanged.

ESC[#P Delete Character (DCH)

Takes one parameter, # (default 1). Deletes # characters starting with the current cursor position. Shifts to the left by # character positions the tail of the current line from the current cursor position inclusive to the end of the line. Blanks are shifted into the rightmost # character positions. The position of the cursor on the screen is unchanged.

ESC[#m Select Graphic Rendition (SGR)

Takes one parameter, # (default 0). Note: unlike most escape sequences, the parameter defaults to zero if omitted. Invokes the graphic rendition specified by the parameter. All following printing characters in the data stream are rendered according to the parameter until the next occurrence of this escape sequence in the data stream. Currently only two graphic renditions are defined:

0 Normal rendition.

7 Negative (reverse) image.

Negative image displays characters as white-on-black if the screen mode is currently black-on white, and vice-versa. Any non-zero value of # is currently equivalent to 7 and selects the negative image rendition.

ESC[p Black On White (SUNBOW)

Takes no parameters. Sets the screen mode to black-on-white. If the screen mode is already black-on-white, has no effect. In this mode spaces display as solid white, other characters as black-on-white. The cursor is a solid black block. Characters displayed in negative image rendition (see 'Select Graphic Rendition' above) is white-on-black in this mode. This is the initial setting of the screen mode on reset.

ESC[q White On Black (SUNWOB)

Takes no parameters. Sets the screen mode to white-on-black. If the screen mode is already white-on-black, has no effect. In this mode spaces display as solid black, other characters as white-on-black. The cursor is a solid white block. Characters displayed in negative image rendition (see 'Select Graphic Rendition' above) is black-on-white in this mode. The initial setting of the screen mode on reset is the alternative mode, black on white.

ESC[#r Set scrolling (SUNSCRL)

Takes one parameter, # (default 0). Sets to # an internal register which determines how many lines the screen scrolls up when a line-feed function is performed with the cursor on the bottom line. A parameter of 2 or 3 introduces a small amount of "jump" when a scroll occurs. A parameter of 34 clears the screen rather than scrolling. The initial setting is 1 on reset.

A parameter of zero initiates "wrap mode" instead of scrolling. In wrap mode, if a linefeed occurs on the bottom line, the cursor goes to the same character position in the top line of the screen. When any linefeed occurs, the line that the cursor moves to is cleared. This means that no scrolling ever occurs. 'ESC [1 r' exits back to scroll mode.

For more information, see the description of the Line-feed (CTRL-J) control function above.

ESC[s Reset terminal emulator (SUNRESET)

Takes no parameters. Resets all modes to default, restores current font from PROM. Screen and cursor position are

4014 TERMINAL EMULATION

The PROM monitor for Sun models 100U and 150U provides the Sun Workstation with the capability to emulate a subset of the Tektronix 4014 terminal. This feature does not exist in other Sun PROMs and will be removed from models 100U and 150U in future Sun releases. **tektool**(1) provides Tektronix 4014 terminal emulation and should be used instead of relying on the capabilities of the PROM monitor.

FILES

/dev/console

SEE ALSO

tektool(1) **kb**(4M), **ldterm**(4M), **pty**(4), **termio**(4), **ttcompat**(4M), **win**(4S), **zs**(4S)

ANSI Standard X3.64, "*Additional Controls for Use with ASCII*", Secretariat: CBEMA, 1828 L St., N.W., Washington, D.C. 20036.

BUGS

TIOCCONS should be restricted to the owner of **/dev/console**.

NAME

db – SunDials STREAMS module

CONFIG

pseudo-device db

SYNOPSIS

```
#include <sys/stream.h>
#include <sundev/vuid_event.h>
#include <sundev/dbio.h>
#include <sys/time.h>
#include <sys/ioctl.h>
open("/dev/dialbox", O_RDWR);
ioctl(fd, I_PUSH, "db");
```

DESCRIPTION

The **db** STREAMS module processes the byte streams generated by the SunDials dial box. The dial box generates a stream of bytes that encode the identity of the dials and the amount by which they are turned.

Each dial sample in the byte stream consists of three bytes. The first byte identifies which dial was turned and the next two bytes return the delta in signed binary format. When bound to an application using the window system, *Virtual User Input Device* events are generated. An event from a dial is constrained to lie between 0x80 and 0x87.

A stream with **db** pushed into it can emit *firm_events* as specified by the protocol of a VUID. **db** understands the **VUIDSFORMAT** and **VUIDGFORMAT** ioctls (see reference below), as defined in **/usr/include/sundev/dbio.h** and **/usr/include/sundev/vuid_event.h**. All other **ioctl()** requests are passed downstream. **db** sets the parameters of a serial port when it is opened. No **termios**(4) **ioctl()** requests should be performed on a **db** STREAMS module, as **db** expects the device parameters to remain as it set them.

IOCTLS

VUIDSFORMAT
VUIDGFORMAT These are standard *Virtual User Input Device* ioctls. See *SunView System Programmer's Guide* for a description of their operation.

FILES

/usr/include/sundev/dbio.h
/usr/include/sundev/vuid_event.h
/usr/include/sys/ioctl.h
/usr/include/sys/stream.h
/usr/include/sys/time.h

SEE ALSO

termios(4), **dialtest**(6), **dbconfig**(8)

SunView System Programmer's Guide,
SunDials Programmers Guide

BUGS

VUIDSADDR and **VUIDGADDR** are not supported.

WARNING

The SunDials dial box must be used with a serial port.

NAME

des – DES encryption chip interface

CONFIG — SUN-3 SYSTEM

device des0 at obio ? csr 0x1c0000

CONFIG — SUN-3x SYSTEM

device des0 at obio ? csr 0x66002000

CONFIG — SUN-4 SYSTEM

device des0 at obio ? csr 0xfe000000

SYNOPSIS

#include <sys/des.h>

DESCRIPTION

The **des** driver provides a high level interface to the AmZ8068 Data Ciphering Processor, a hardware implementation of the NBS Data Encryption Standard.

The high level interface provided by this driver is hardware independent and could be shared by future drivers in other systems.

The interface allows access to two modes of the DES algorithm: Electronic Code Book (ECB) and Cipher Block Chaining (CBC). All access to the DES driver is through **ioctl**(2) calls rather than through reads and writes; all encryption is done in-place in the user's buffers.

IOCTLS

The ioctls provided are:

DESIOCBLOCK

This call encrypts/decrypts an entire buffer of data, whose address and length are passed in the '**struct desparams**' addressed by the argument. The length must be a multiple of 8 bytes.

DESIOCQUICK

This call encrypts/decrypts a small amount of data quickly. The data is limited to **DES_QUICKLEN** bytes, and must be a multiple of 8 bytes. Rather than being addresses, the data is passed directly in the '**struct desparams**' argument.

FILES

/dev/des

SEE ALSO

des(1), **des_crypt**(3)

Federal Information Processing Standards Publication 46

AmZ8068 DCP Product Description, Advanced Micro Devices

NAME

dkio – generic disk control operations

DESCRIPTION

All Sun disk drivers support a set of **ioctl**(2) requests for disk formatting and labeling operations. Basic to these **ioctl**( ) requests are the definitions in **/usr/include/sun/dkio.h**:

```
/*
* Structures and definitions for disk I/O control commands
*/
/* Controller and disk identification */
struct dk_info {
        int     dki_ctlr;               /* controller address */
        short   dki_unit;               /* unit (slave) address */
        short   dki_ctype;              /* controller type */
        short   dki_flags;              /* flags */
};
/* controller types */
#define DKC_UNKNOWN             0
#define DKC_DSD5215             5
#define DKC_XY450               6
#define DKC_ACB4000             7
#define DKC_MD21                8
#define DKC_XD7053              11
#define DKC_CSS                 12
#define DKC_NEC765              13      /* floppy on Sun386i */
#define DKC_INTEL82072          14
/* flags */
#define  DKI_BAD144     0x01    /* use DEC std 144 bad sector fwding */
#define  DKI_MAPTRK     0x02    /* controller does track mapping */
#define  DKI_FMTTRK     0x04    /* formats only full track at a time */
#define  DKI_FMTVOL     0x08    /* formats only full volume at a time */
/* Definition of a disk's geometry */
struct dk_geom {
        unsigned short  dkg_ncyl;       /* # of data cylinders */
        unsigned short  dkg_acyl;       /* # of alternate cylinders */
        unsigned short  dkg_bcyl;       /* cyl offset (for fixed head area) */
        unsigned short  dkg_nhead;      /* # of heads */
        unsigned short  dkg_bhead;      /* head offset (for Larks, etc.) */
        unsigned short  dkg_nsect;      /* # of sectors per track */
        unsigned short  dkg_intrlv;     /* interleave factor */
        unsigned short  dkg_gap1;       /* gap 1 size */
        unsigned short  dkg_gap2;       /* gap 2 size */
        unsigned short  dkg_apc;        /* alternates per cyl (SCSI only) */
        unsigned short  dkg_extra[9];   /* for compatible expansion */
};
/* Partition map (part of dk_label) */
struct dk_map {
        long    dkl_cylno;      /* starting cylinder */
        long    dkl_nblk;       /* number of blocks */
};
```

```
/* Floppy characteristics */
struct fdk_char {
        u_char  medium;         /* medium type (scsi floppy only) */
        int     transfer_rate;  /* transfer rate */
        int     ncyl;           /* number of cylinders */
        int     nhead;          /* number of heads */
        int     sec_size;       /* sector size */
        int     secptrack;      /* sectors per track */
        int     steps;          /* number of steps per */
};
/* Used by FDKGETCHANGE, returned state of the sense disk change bit. */
#define FDKGC_HISTORY       0x01    /* disk has changed since last call */
#define FDKGC_CURRENT       0x02    /* current state of disk change */
/* disk I/O control commands */
#define  DKIOCINFO          _IOR(d, 8, struct dk_info)      /* Get info */
#define  DKIOCGGEOM         _IOR(d, 2, struct dk_geom)      /* Get geometry */
#define  DKIOCSGEOM         _IOW(d, 3, struct dk_geom)      /* Set geometry */
#define  DKIOCGPART         _IOR(d, 4, struct dk_map)       /* Get partition info */
#define  DKIOCSPART         _IOW(d, 5, struct dk_map)       /* Set partition info */
#define  DKIOCWCHK          _IOWR(d, 115, int)              /* Toggle write check */
/* floppy I/O control commands */
#define  FDKIOGCHAR         _IOR(d, 114, struct fdk_char)   /* Get floppy characteristics */
#define  FDKEJECT           _IO(d, 112)                     /* Eject floppy */
#define  FDKGETCHANGE       _IOR(d, 111, int)               /* Get disk change status */
```

The DKIOCINFO ioctl returns a dk_info structure which tells the type of the controller and attributes about how bad-block processing is done on the controller. The DKIOCGPART and DKIOCSPART get and set the controller's current notion of the partition table for the disk (without changing the partition table on the disk itself), while the DKIOCGGEOM and DKIOCSGEOM ioctls do similar things for the per-drive geometry information. The DKIOCWCHK enables or disables a disk's write check capabilities. The FDKIOGCHAR ioctl returns an fdk_char structure which gives the characteristics of the floppy diskette. The FDKEJECT ioctl ejects the floppy diskette. The FDKGETCHANGE returns the status of the diskette changed signal from the floppy interface.

FILES

/usr/include/sun/dkio.h

SEE ALSO

fd(4S), ip(4P), sd(4S), xd(4S), xy(4S), dkctl(8)

NAME

drum – paging device

CONFIG

None; included with standard system.

SYNOPSIS

#include <fcntl.h>

open("/dev/drum", *mode***);**

DESCRIPTION

This file refers to the paging device in use by the system. This may actually be a subdevice of one of the disk drivers, but in a system with paging interleaved across multiple disk drives it provides an indirect driver for the multiple drives.

FILES

/dev/drum

BUGS

Reads from the drum are not allowed across the interleaving boundaries. Since these only occur every .5Mbytes or so, and since the system never allocates blocks across the boundary, this is usually not a problem.

NAME

fb – driver for Sun console frame buffer

CONFIG

None; included in standard system.

DESCRIPTION

The **fb** driver provides indirect access to a Sun frame buffer. It is an indirect driver for the Sun workstation console's frame buffer. At boot time, the workstation's frame buffer device is determined from information from the PROM monitor and set to be the one that **fb** will indirect to. The device driver for the console's frame buffer must be configured into the kernel so that this indirect driver can access it.

The idea behind this driver is that user programs can open a known device, query its characteristics and access it in a device dependent way, depending on the type. **fb** redirects **open**(2V), **close**(2V), **ioctl**(2), and **mmap**(2) calls to the real frame buffer. All Sun frame buffers support the same general interface; see **fbio**(4S).

FILES

/dev/fb

SEE ALSO

close(2V), **ioctl**(2), **mmap**(2), **open**(2V), **fbio**(4S)

NAME

fbio – frame buffer control operations

DESCRIPTION

All Sun frame buffers support the same general interface that is defined by <sun/fbio.h>. Each responds to an **FBIOGTYPE** **ioctl(2)** request which returns information in a **fbtype** structure.

Each device has an **FBTYPE** which is used by higher-level software to determine how to perform graphics functions. Each device is used by opening it, doing an **FBIOGTYPE** **ioctl()** to see which frame buffer type is present, and thereby selecting the appropriate device-management routines.

Full-fledged frame buffers (that is, those that run SunView1) implement an **FBIOGPIXRECT** **ioctl()** request, which returns a pixrect. This call is made only from inside the kernel. The returned pixrect is used by **win**(4S) for cursor tracking and colormap loading.

FBIOSVIDEO and **FBIOGVIDEO** are general-purpose **ioctl()** reuqests for controlling possible video features of frame buffers. These **ioctl()** requests either set or return the value of a flags integer. At this point, only the **FBVIDEO_ON** option is available, controlled by **FBIOSVIDEO**. **FBIOGVIDEO** returns the current video state.

The **FBIOSATTR** and **FBIOGATTR** **ioctl()** requests allow access to special features of newer frame buffers. They use the **fbsattr** and **fbgattr** structures.

Some color frame buffers support the **FBIOPUTCMAP** and **FBIOGETCMAP** **ioctl()** reuqests, which provide access to the colormap. They use the **fbcmap** structure.

SEE ALSO

ioctl(2), **mmap**(2), **bw***(4S), **cg***(4S), **gp***(4S), **fb**(4S), **win**(4S)

BUGS

The **FBIOSATTR** and **FBIOGATTR** **ioctl()** requests are only supported by frame buffers which emulate older frame buffer types. For example, **cgfour**(4S) frame buffers emulate **bwtwo**(4S) frame buffers. If a frame buffer is emulating another frame buffer, **FBIOGTYPE** returns the emulated type. To get the real type, use **FBIOGATTR**.

NAME

fd – disk driver for Floppy Disk Controllers

CONFIG — Sun386i SYSTEMS

controller fdc0 at atmem ? csr 0x1000 dmachan 2 irq 6 priority 2
disk fd0 at fdc0 drive 0 flags 0

CONFIG — SUN-3/80 SYSTEMS

controller fdc0 at obio ? csr 0x6e000000 priority 6 vector fdintr 0x5c
disk fd0 at fdc0 drive 0 flags 0

CONFIG — SPARCstation 1 SYSTEMS

device-driver fd

AVAILABILITY

Sun386i, Sun-3/80, and SPARCstation 1 systems only.

DESCRIPTION

The **fd** driver provides an interface to floppy disks using the Intel 82072 disk controller on Sun386i, Sun-3/80 and SPARCstation 1 systems.

The minor device number in files that use the floppy interface encodes the unit number as well as the partition. The bits of the minor device number are defined as **rrruuppp** where **r**=reserved, **u**=unit, and **p**=partition. The unit number selects a particular floppy drive for the controller. The partition number picks one of eight partitions [**a-h**].

When the floppy is first opened the driver looks for a label in logical block 0 of the diskette. If a label is found, the geometry and partition information from the label will be used on each access thereafter. The driver first assumes high density characteristics when it tries to read the label. If the read fails it will try the read again using low density characteristics. If both attempts to read the label fail, the open will fail. Use the **FNDELAY** flag when opening an unformatted diskette as a signal to the driver that it should not attempt to access the diskette. If block 0 is read successfully, but a label is not found, the open will fail for the block interface. Using the raw interface, the open will succeed even if the diskette is unlabeled. Default geometry and partitioning are assumed if the diskette is unlabeled.

The default partitions are:

a –> 0, N-1

b –> N-1, N

c –> 0, N

where N is the number of cylinders on the diskette.

The **fd** driver supports both block and raw interfaces. The block files access the disk using the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a "raw" interface that provides for direct transmission between the disk and the user's read or write buffer. A single **read**(2V) or **write**(2V) call usually results in one I/O operation; therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw files conventionally begin with an extra '**r**'.

FILES — Sun386i SYSTEMS

1.44 MB Floppy Disk Drives:

/dev/fd0a	block file
/dev/fd0c	block file
/dev/rfd0a	raw file
/dev/rfd0c	raw file

720 K Floppy Disk Drives:

/dev/fd1a	block file
/dev/fd1c	block file
/dev/rfd1a	raw file
/dev/rfd1c	raw file

FILES — SUN-3/80 and SPARCstation 1 SYSTEMS

Note: the **fd** driver on Sun-3/80 and SPARCstation 1 systems auto-senses the density of the floppy.

/dev/fd0[a-c]	block file
/dev/fd0	block file (same as **/dev/fd0c**)
/dev/rfd0[a-c]	raw file
/dev/rfd0	raw file (same as **/dev/rfd0c**)

SEE ALSO

read(2V), **write**(2V), **dkio**(4S)

DIAGNOSTICS — Sun386i SYSTEMS

fd drv %d, trk %d: %s

A command such as read or write encountered a format-related error condition. The value of *%s* is derived from the error number given by the controller, indicating the nature of the error. The track number is relative to the beginning of the partition involved.

fd drv %d, blk %d: %s

A command such as read or write encountered an error condition related to I/O. The value of *%s* is derived from the error number returned by the controller and indicates the nature of the error. The block number is relative to the start of the partition involved.

fd controller: %s

An error occurred in the controller. The value of *%s* is derived from the status returned by the controller and specifies the error encountered.

fd(%d): %s please insert

I/O was attempted while the floppy drive door was not latched. The value of *%s* indicates which disk was expected to be in the drive.

DIAGNOSTICS — SUN-3/80 and SPARCstation 1 SYSTEMS

fd%d: %s failed (%x %x %x)

The command, %s, failed after several retries on drive %d. The three hex values in parenthesis are the contents of status register 0, status register 1, and status register 2 of the Intel 82072 Floppy Disk Controller on completion of the command as documented in the data sheet for that part. This error message is usually followed by one of the following, interpreting the bits of the status register:

fd%d: not writable
fd%d: crc error
fd%d: overrun/underrun
fd%d: bad format
fd%d: timeout

NOTES

Floppy diskettes have 18 sectors per track, and can cross a track (though not a cylinder) boundary without lossing data, so when using **dd**(1) to or from a diskette, you should specify **bs=18k** or multiples thereof.

NAME

filio – ioctls that operate directly on files, file descriptors, and sockets

SYNOPSIS

#include <sys/filio.h>

DESCRIPTION

The IOCTL's listed in this manual page apply directly to files, file descriptors, and sockets, independent of any underlying device or protocol.

Note: the **fcntl**(2V) system call is the primary method for operating on file descriptors as such, rather than on the underlying files.

IOCTLS for File Descriptors

FIOCLEX The argument is ignored. Set the close-on-exec flag for the file descriptor passed to **ioctl**. This flag is also manipulated by the **F_SETFD** command of **fcntl**(2V).

FIONCLEX The argument is ignored. Clear the close-on-exec flag for the file descriptor passed to **ioctl**.

IOCTLs for Files

FIONREAD The argument is a pointer to a **long**. Set the value of that **long** to the number of immediately readable characters from whatever the descriptor passed to **ioctl** refers to. This works for files, pipes, sockets, and terminals.

FIONBIO The argument is a pointer to an **int**. Set or clear non-blocking I/O. If the value of that **int** is a 1 (one) the descriptor is set for non-blocking I/O. If the value of that **int** is a 0 (zero) the descriptor is cleared for non-blocking I/O.

FIOASYNC The argument is a pointer to an **int**. Set or clear asynchronous I/O. If the value of that **int** is a 1 (one) the descriptor is set for asynchronous I/O. If the value of that **int** is a 0 (zero) the descriptor is cleared for asynchronous I/O.

FIOSETOWN The argument is a pointer to an **int**. Set the process-group ID that will subsequently receive **SIGIO** or **SIGURG** signals for the object referred to by the descriptor passed to **ioctl** to the value of that **int**.

FIOGETOWN The argument is a pointer to an **int**. Set the value of that **int** to the process-group ID that is receiving **SIGIO** or **SIGURG** signals for the object referred to by the descriptor passed to **ioctl**.

SEE ALSO

ioctl(2), **fcntl**(2V), **getsockopt**(2), **sockio**(4)

NAME

fpa – Sun-3/Sun-3x floating-point accelerator

CONFIG — SUN-3/SUN-3X SYSTEMS

device fpa0 at virtual ? csr 0xe0000000

SYNOPSIS

#include <sundev/fpareg.h>
open("/dev/fpa", flags);

DESCRIPTION

FPA and FPA+ are compatible floating point accelerators available on certain Sun-3 and Sun-3x systems. They provide hardware contexts for simultaneous use by up to 32 processes. The same **fpa** device driver manages either FPA or FPA+ hardware.

Processes access the device using **open**(2V) and **close**(2V) system calls, and the FPA is automatically mapped into the process' address space by SunOS. This is normally provided transparently at compile time by a compiler option, such as the **–ffpa** option to **cc**(1V).

The valid **ioctl**(2) system calls are used only by diagnostics and by system administration programs, such as **fpa_download**(8).

IOCTLS

FPA_ACCESS_OFF	Clear **FPA_ACCESS_BIT** in FPA state register to disable access to constants RAM using FPA load pointer.
FPA_ACCESS_ON	Set **FPA_ACCESS_BIT** in FPA state register to enable access to constants RAM using FPA load pointer.
FPA_FAIL	Disable the FPA.
FPA_GET_DATAREGS	Return the contents of 8 FPA registers.
FPA_INIT_DONE	Called when downloading is complete. Allows multiple users to access the FPA.
FPA_LOAD_OFF	Set **FPA_LOAD_BIT** in FPA state register to disable access to microstore or map RAM via FPA load pointer.
FPA_LOAD_ON	Set **FPA_LOAD_BIT** in FPA state register to enable access to microstore or map RAM using FPA load pointer.

The following two **ioctl**() requests are for diagnostic use only. **fpa** must be compiled with **FPA_DIAGNOSTICS_ONLY** defined to enable these two calls.

FPA_WRITE_STATE	Overwrite the FPA state register.
FPA_WRITE_HCP	Write to the hard clear pipe register.

ERRORS

The following error messages are returned by **open** system calls only.

EBUSY	All 32 FPA contexts are being used.
EEXIST	The current process has already opened **/dev/fpa**.
EIO	Downloading has not completed, so only 1 root process can have the FPA open at a time.
ENETDOWN	FPA is disabled.
ENOENT	68881 chip does not exist.
ENXIO	FPA board does not exist.

The following error messages are returned by **ioctl** system calls only.

EINVAL	Invalid ioctl. This may occur if diagnostic only ioctls, **FPA_WRITE_STATE** or **FPA_WRITE_HCP**, are used with a driver which didn't compile in those calls.

EPERM All ioctl calls except for **FPA_GET_DATAREGS** require root execution level.

EPIPE The FPA pipe is not clear.

FILES

/dev/fpa device file for both FPA and FPA+.

SEE ALSO

cc(1V), **close**(2V), **ioctl**(2), **open**(2V) **fpa_download**(8), **fparel**(8), **fpaversion**(8)

DIAGNOSTICS

If hardware problems are detected then all processes with **/dev/fpa** open are killed, and future opens of **/dev/fpa** are disabled.

NAME

gpone – graphics processor

CONFIG — SUN-3, SUN-3x, SUN-4 SYSTEMS

```
device gpone0 at vme24d16 ? csr 0x210000        # GP or GP+
device gpone0 at vme24d32 ? csr 0x240000        # GP2
```

DESCRIPTION

The **gpone** interface provides access to the optional Graphics Processor Board (GP).

The hardware consumes 64 kilobytes of VME bus address space. The GP board starts at standard address 0x210000 and must be configured for interrupt level 4.

IOCTLS

The graphics processor responds to a number of ioctl calls as described here. One of the calls uses a **gp1fbinfo** structure that looks like this:

```
struct  gp1fbinfo {
        int             fb_vmeaddr;     /* physical color board address */
        int             fb_hwwidth;     /* fb board width */
        int             fb_hwheight;    /* fb board height */
        int             addrdelta;      /* phys addr diff between fb and gp */
        caddr_t         fb_ropaddr;     /* cg2 va thru kernelmap */
        int             fbunit;         /* fb unit to use  for a,b,c,d */
};
```

The ioctl call looks like this:

```
ioctl(file, request, argp)
int file, request;
```

argp is defined differently for each GP ioctl request and is specified in the descriptions below.

The following ioctl commands provide for transferring data between the graphics processor and color boards and processes.

GP1IO_PUT_INFO
Passes information about the frame buffer into driver. **argp** points to a **struct gp1fbinfo** which is passed to the driver.

GP1IO_GET_STATIC_BLOCK
Hands out a static block from the GP. **argp** points to an **int** which is returned from the driver.

GP1IO_FREE_STATIC_BLOCK
Frees a static block from the GP. **argp** points to an **int** which is passed to the driver.

GP1IO_GET_GBUFFER_STATE
Checks to see if there is a buffer present on the GP. **argp** points to an **int** which is returned from the driver.

GP1IO_CHK_GP
Restarts the GP if necessary. **argp** points to an **int** which is passed to the driver.

GP1IO_GET_RESTART_COUNT
Returns the number of restarts of a GP since power on. Needed to differentiate **SIGXCPU** calls in user processes. **argp** points to an **int** which is returned from the driver.

GP1IO_REDIRECT_DEVFB
Configures **/dev/fb** to talk to a graphics processor device. **argp** points to an **int** which is passed to the driver.

GP1IO_GET_REQDEV
Returns the requested minor device. **argp** points to a **dev_t** which is returned from the driver.

GP1IO_GET_TRUMINORDEV

Returns the true minor device. **argp** points to a **char** which is returned from the driver.

The graphics processor driver also responds to the **FBIOGTYPE**, ioctl which a program can use to inquire as to the characteristics of the display device, the **FBIOGINFO**, ioctl for passing generic information, and the **FBIOGPIXRECT** ioctl so that SunWindows can run on it. See **fbio**(4S).

FILES

/dev/fb
/dev/gpone[0-3][abcd]

SEE ALSO

fbio(4S), **mmap**(2), **gpconfig**(8)

SunCGI Reference Manual

DIAGNOSTICS

The Graphics Processor has been restarted. You may see display garbage as a result.

NAME

icmp – Internet Control Message Protocol

SYNOPSIS

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

s = socket(AF_INET, SOCK_RAW, proto);
```

DESCRIPTION

ICMP is the error and control message protocol used by the Internet protocol family. It is used by the kernel to handle and report errors in protocol processing. It may also be accessed through a "raw socket" for network monitoring and diagnostic functions. The protocol number for ICMP, used in the *proto* parameter to the socket call, can be obtained from **getprotobyname** (see **getprotoent**(3N)). ICMP sockets are connectionless, and are normally used with the *sendto* and *recvfrom* calls, though the **connect**(2) call may also be used to fix the destination for future packets (in which case the **read**(2V) or **recv**(2) and **write**(2V) or **send**(2) system calls may be used).

Outgoing packets automatically have an Internet Protocol (IP) header prepended to them. Incoming packets are provided to the holder of a raw socket with the IP header and options intact.

ICMP is an unreliable datagram protocol layered above IP. It is used internally by the protcol code for various purposes including routing, fault isolation, and congestion control. Receipt of an ICMP "redirect" message will add a new entry in the routing table, or modify an existing one. ICMP messages are routinely sent by the protocol code. Received ICMP messages may be reflected back to users of higher-level protocols such as TCP or UDP as error returns from system calls. A copy of all ICMP message received by the system is provided using the ICMP raw socket.

ERRORS

A socket operation may fail with one of the following errors returned:

EISCONN	when trying to establish a connection on a socket which already has one, or when trying to send a datagram with the destination address specified and the socket is already connected;
ENOTCONN	when trying to send a datagram, but no destination address is specified, and the socket hasn't been connected;
ENOBUFS	when the system runs out of memory for an internal data structure;
EADDRNOTAVAIL	when an attempt is made to create a socket with a network address for which no network interface exists.

SEE ALSO

connect(2), **read**(2V), **recv**(2), **send**(2), **write**(2V), **getprotoent**(3N), **inet**(4F), **ip**(4P), **routing**(4N)

Postel, Jon, *Internet Control Message Protocol — DARPA Internet Program Protocol Specification*, RFC 792, Network Information Center, SRI International, Menlo Park, Calif., September 1981. (Sun 800-1064-01)

BUGS

Replies to ICMP "echo" messages which are source routed are not sent back using inverted source routes, but rather go back through the normal routing mechanisms.

NAME

ie – Intel 10 Mb/s Ethernet interface

CONFIG — SUN-4 SYSTEM

device ie0 at obio ? csr 0x6000000 priority 3
device ie1 at vme24d16 ? csr 0xe88000 priority 3 vector ieintr 0x75
device ie2 at vme24d16 ? csr 0x31ff02 priority 3 vector ieintr 0x76
device ie3 at vme24d16 ? csr 0x35ff02 priority 3 vector ieintr 0x77

CONFIG — SUN-3x SYSTEM

device ie0 at obio ? csr 0x65000000 priority 3
device ie1 at vme24d16 ? csr 0xe88000 priority 3 vector ieintr 0x75
device ie2 at vme24d32 ? csr 0x31ff02 priority 3 vector ieintr 0x76
device ie3 at vme24d32 ? csr 0x35ff02 priority 3 vector ieintr 0x77

CONFIG — SUN-3 SYSTEM

device ie0 at obio ? csr 0xc0000 priority 3
device ie1 at vme24d16 ? csr 0xe88000 priority 3 vector ieintr 0x75
device ie2 at vme24d32 ? csr 0x31ff02 priority 3 vector ieintr 0x76
device ie3 at vme24d32 ? csr 0x35ff02 priority 3 vector ieintr 0x77

CONFIG — SUN-3E SYSTEM

device ie0 at vme24d16 ? csr 0x31ff02 priority 3 vector ieintr 0x74

CONFIG — SUN386i SYSTEM

device ie0 at obmem ? csr 0xD0000000 irq 21 priority 3

DESCRIPTION

The ie interface provides access to a 10 Mb/s Ethernet network through a controller using the Intel 82586 LAN Coprocessor chip. For a general description of network interfaces see if(4N).

ie0 specifies a CPU-board-resident interface, except on a Sun-3E where **ie0** is the Sun-3/E Ethernet expansion board. **ie1** specifies a Multibus Intel Ethernet interface for use with a VME adapter. **ie2** and **ie3** specify SunNet Ethernet/VME Controllers, also known as a Sun-3/E Ethernet expansion boards.

SEE ALSO

if(4N), le(4S)

DIAGNOSTICS

There are too many driver messages to list them all individually here. Some of the more common messages and their meanings follow.

ie%d: Ethernet jammed
Network activity has become so intense that sixteen successive transmission attempts failed, and the 82586 gave up on the current packet. Another possible cause of this message is a noise source somewhere in the network, such as a loose transceiver connection.

ie%d: no carrier
The 82586 has lost input to its carrier detect pin while trying to transmit a packet, causing the packet to be dropped. Possible causes include an open circuit somewhere in the network and noise on the carrier detect line from the transceiver.

ie%d: lost interrupt: resetting
The driver and 82586 chip have lost synchronization with each other. The driver recovers by resetting itself and the chip.

ie%d: iebark reset
The 82586 failed to complete a watchdog timeout command in the allotted time. The driver recovers by resetting itself and the chip.

ie%d: WARNING: requeuing

The driver has run out of resources while getting a packet ready to transmit. The packet is put back on the output queue for retransmission after more resources become available.

ie%d: panic: scb overwritten

The driver has discovered that memory that should remain unchanged after initialization has become corrupted. This error usually is a symptom of a bad 82586 chip.

ie%d: giant packet

Provided that all stations on the Ethernet are operating according to the Ethernet specification, this error "should never happen," since the driver allocates its receive buffers to be large enough to hold packets of the largest permitted size. The most likely cause of this message is that some other station on the net is transmitting packets whose lengths exceed the maximum permitted for Ethernet.

NAME

if – general properties of network interfaces

DESCRIPTION

Each network interface in a system corresponds to a path through which messages may be sent and received. A network interface usually has a hardware device associated with it, though certain interfaces such as the loopback interface, **lo**(4), do not.

At boot time, each interface with underlying hardware support makes itself known to the system during the autoconfiguration process. Once the interface has acquired its address, it is expected to install a routing table entry so that messages can be routed through it. Most interfaces require some part of their address specified with an SIOCSIFADDR IOCTL before they will allow traffic to flow through them. On interfaces where the network-link layer address mapping is static, only the network number is taken from the ioctl; the remainder is found in a hardware specific manner. On interfaces which provide dynamic network-link layer address mapping facilities (for example, 10Mb/s Ethernets using **arp**(4P)), the entire address specified in the ioctl is used.

The following **ioctl** calls may be used to manipulate network interfaces. Unless specified otherwise, the request takes an **ifreq** structure as its parameter. This structure has the form

```
struct  ifreq {
        char    ifr_name[16];           /* name of interface (e.g. "ec0") */
        union {
                struct  sockaddr ifru_addr;
                struct  sockaddr ifru_dstaddr;
                short   ifru_flags;
        } ifr_ifru;
#define ifr_addr                ifr_ifru.ifru_addr      /* address */
#define ifr_dstaddr     ifr_ifru.ifru_dstaddr   /* other end of p-to-p link */
#define ifr_flags               ifr_ifru.ifru_flags     /* flags */
};
```

SIOCSIFADDR Set interface address. Following the address assignment, the "initialization" routine for the interface is called.

SIOCGIFADDR Get interface address.

SIOCSIFDSTADDR Set point to point address for interface.

SIOCGIFDSTADDR Get point to point address for interface.

SIOCSIFFLAGS Set interface flags field. If the interface is marked down, any processes currently routing packets through the interface are notified.

SIOCGIFFLAGS Get interface flags.

SIOCGIFCONF Get interface configuration list. This request takes an **ifconf** structure (see below) as a value-result parameter. The **ifc_len** field should be initially set to the size of the buffer pointed to by **ifc_buf**. On return it will contain the length, in bytes, of the configuration list.

```
/*
* Structure used in SIOCGIFCONF request.
* Used to retrieve interface configuration
* for machine (useful for programs which
* must know all networks accessible).
*/
struct  ifconf {
        int     ifc_len;        /* size of associated buffer */
        union {
                caddr_t ifcu_buf;
                struct  ifreq *ifcu_req;
        } ifc_ifcu;
#define ifc_buf  ifc_ifcu.ifcu_buf /* buffer address */
#define ifc_req  ifc_ifcu.ifcu_req /* array of structures returned */
};
```

SIOCADDMULTI Enable a multicast address for the interface. A maximum of 64 multicast addresses may be enabled for any given interface.

SIOCDELMULTI Disable a previously set multicast address.

SIOCSPROMISC Toggle promiscuous mode.

SEE ALSO

arp(4P), **lo**(4)

NAME

inet – Internet protocol family

SYNOPSIS

options INET

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

DESCRIPTION

The Internet protocol family implements a collection of protocols which are centered around the *Internet Protocol* (IP) and which share a common address format. The Internet family provides protocol support for the SOCK_STREAM, SOCK_DGRAM, and SOCK_RAW socket types.

PROTOCOLS

The Internet protocol family is comprised of the Internet Protocol (IP), the Address Resolution Protocol (ARP), the Internet Control Message Protocol (ICMP), the Transmission Control Protocol (TCP), and the User Datagram Protocol (UDP).

TCP is used to support the SOCK_STREAM abstraction while UDP is used to support the SOCK_DGRAM abstraction; see **tcp**(4P) and **udp**(4P). A raw interface to IP is available by creating an Internet socket of type SOCK_RAW; see **ip**(4P). ICMP is used by the kernel to handle and report errors in protocol processing. It is also accessible to user programs; see **icmp**(4P). ARP is used to translate 32-bit IP addresses into 48-bit Ethernet addresses; see **arp**(4P).

The 32-bit IP address is divided into network number and host number parts. It is frequency-encoded; the most-significant bit is zero in Class A addresses, in which the high-order 8 bits are the network number. Class B addresses have their high order two bits set to 10 and use the high-order 16 bits as the network number field. Class C addresses have a 24-bit network number part of which the high order three bits are 110. Sites with a cluster of local networks may chose to use a single network number for the cluster; this is done by using subnet addressing. The local (host) portion of the address is further subdivided into subnet number and host number parts. Within a subnet, each subnet appears to be an individual network; externally, the entire cluster appears to be a single, uniform network requiring only a single routing entry. Subnet addressing is enabled and examined by the following **ioctl**(2) commands on a datagram socket in the Internet domain; they have the same form as the SIOCIFADDR command (see **intro**(4)).

SIOCSIFNETMASK — Set interface network mask. The network mask defines the network part of the address; if it contains more of the address than the address type would indicate, then subnets are in use.

SIOCGIFNETMASK — Get interface network mask.

ADDRESSING

IP addresses are four byte quantities, stored in network byte order (on Sun386i systems these are word and byte reversed).

Sockets in the Internet protocol family use the following addressing structure:

```
struct sockaddr_in {
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};
```

Library routines are provided to manipulate structures of this form; see **intro**(3).

The **sin_addr** field of the **sockaddr_in** structure specifies a local or remote IP address. Each network interface has its own unique IP address. The special value INADDR_ANY may be used in this field to effect "wildcard" matching. Given in a **bind**(2) call, this value leaves the local IP address of the socket unspecified, so that the socket will receive connections or messages directed at any of the valid IP addresses of the system. This can prove useful when a process neither knows nor cares what the local IP

address is or when a process wishes to receive requests using all of its network interfaces. The **sockaddr_in** structure given in the **bind**(2) call must specify an **in_addr** value of either **IPADDR_ANY** or one of the system's valid IP addresses. Requests to bind any other address will elicit the error EADDRNOTAVAIL. When a **connect**(2) call is made for a socket that has a wildcard local address, the system sets the **sin_addr** field of the socket to the IP address of the network interface that the packets for that connection are routed via.

The **sin_port** field of the **sockaddr_in** structure specifies a port number used by TCP or UDP. The local port address specified in a **bind**(2) call is restricted to be greater than **IPPORT_RESERVED** (defined in **<netinet/in.h>**) unless the creating process is running as the super-user, providing a space of protected port numbers. In addition, the local port address must not be in use by any socket of same address family and type. Requests to bind sockets to port numbers being used by other sockets return the error EADDRINUSE. If the local port address is specified as 0, then the system picks a unique port address greater than **IPPORT_RESERVED**. A unique local port address is also picked when a socket which is not bound is used in a **connect**(2) or **send**(2) call. This allows programs which do not care which local port number is used to set up TCP connections by simply calling **socket**(2) and then **connect**(2), and to send UDP datagrams with a **socket**(2) call followed by a **send**(2) call.

Although this implementation restricts sockets to unique local port numbers, TCP allows multiple simultaneous connections involving the same local port number so long as the remote IP addresses or port numbers are different for each connection. Programs may explicitly override the socket restriction by setting the **SO_REUSEADDR** socket option with **setsockopt** (see **getsockopt**(2)).

SEE ALSO

bind(2), **connect**(2), **getsockopt**(2), **ioctl**(2), **send**(2), **socket**(2), **intro**(3), **byteorder**(3N), **gethostent**(3N), **getnetent**(3N), **getprotoent**(3N), **getservent**(3N), **inet**(3N), **intro**(4), **arp**(4P), **icmp**(4P), **ip**(4P) **tcp**(4P), **udp**(4P)

Network Information Center, *DDN Protocol Handbook* (3 vols.), Network Information Center, SRI International, Menlo Park, Calif., 1985.
A 4.2BSD Interprocess Communication Primer

WARNING

The Internet protocol support is subject to change as the Internet protocols develop. Users should not depend on details of the current implementation, but rather the services exported.

NAME

ip – Internet Protocol

SYNOPSIS

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

s = socket(AF_INET, SOCK_RAW, proto);
```

DESCRIPTION

IP is the internetwork datagram delivery protocol that is central to the Internet protocol family. Programs may use IP through higher-level protocols such as the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP), or may interface directly using a "raw socket." See **tcp**(4P) and **udp**(4P). The protocol options defined in the IP specification may be set in outgoing datagrams.

Raw IP sockets are connectionless and are normally used with the **sendto** and **recvfrom** calls, (see **send**(2) and **recv**(2)) although the **connect**(2) call may also be used to fix the destination for future datagrams (in which case the **read**(2V) or **recv**(2) and **write**(2V) or **send**(2) calls may be used). If **proto** is zero, the default protocol, **IPPROTO_RAW**, is used. If **proto** is non-zero, that protocol number will be set in outgoing datagrams and will be used to filter incoming datagrams. An IP header will be generated and prepended to each outgoing datagram; Received datagrams are returned with the IP header and options intact.

A single socket option, **IP_OPTIONS**, is supported at the IP level. This socket option may be used to set IP options to be included in each outgoing datagram. IP options to be sent are set with **setsockopt** (see **getsockopt**(2)). The **getsockopt**(2) call returns the IP options set in the last **setsockopt** call. IP options on received datagrams are visible to user programs only using raw IP sockets. The format of IP options given in **setsockopt** matches those defined in the IP specification with one exception: the list of addresses for the source routing options must include the first-hop gateway at the beginning of the list of gateways. The first-hop gateway address will be extracted from the option list and the size adjusted accordingly before use. IP options may be used with any socket type in the Internet family.

At the socket level, the socket option **SO_DONTROUTE** may be applied. This option forces datagrams being sent to bypass the routing step in output. Normally, IP selects a network interface to send the datagram via, and possibly an intermediate gateway, based on an entry in the routing table. See **routing**(4N). When **SO_DONTROUTE** is set, the datagram will be sent via the interface whose network number or full IP address matches the destination address. If no interface matches, the error ENETUNRCH will be returned.

Datagrams flow through the IP layer in two directions: from the network **ip** to user processes and from user processes *down* to the network. Using this orientation, IP is layered *above* the network interface drivers and *below* the transport protocols such as UDP and TCP. The Internet Control Message Protocol (ICMP) is logically a part of IP. See **icmp**(4P).

IP provides for a checksum of the header part, but not the data part of the datagram. The checksum value is computed and set in the process of sending datagrams and checked when receiving datagrams. IP header checksumming may be disabled for debugging purposes by patching the kernel variable **ipcksum** to have the value zero.

IP options in received datagrams are processed in the IP layer according to the protocol specification. Currently recognized IP options include: security, loose source and record route (LSRR), strict source and record route (SSRR), record route, stream identifier, and internet timestamp.

The IP layer will normally forward received datagrams that are not addressed to it. Forwarding is under the control of the kernel variable *ipforwarding*: if *ipforwarding* is zero, IP datagrams will not be forwarded; if *ipforwarding* is one, IP datagrams will be forwarded. *ipforwarding* is usually set to one only in machines with more than one network interface (internetwork routers). This kernel variable can be patched to enable or disable forwarding.

The IP layer will send an ICMP message back to the source host in many cases when it receives a datagram that can not be handled. A "time exceeded" ICMP message will be sent if the "time to live" field in the IP header drops to zero in the process of forwarding a datagram. A "destination unreachable" message will be sent if a datagram can not be forwarded because there is no route to the final destination, or if it can not be fragmented. If the datagram is addressed to the local host but is destined for a protocol that is not supported or a port that is not in use, a destination unreachable message will also be sent. The IP layer may send an ICMP "source quench" message if it is receiving datagrams too quickly. ICMP messages are only sent for the first fragment of a fragmented datagram and are never returned in response to errors in other ICMP messages.

The IP layer supports fragmentation and reassembly. Datagrams are fragmented on output if the datagram is larger than the maximum transmission unit (MTU) of the network interface. Fragments of received datagrams are dropped from the reassembly queues if the complete datagram is not reconstructed within a short time period.

Errors in sending discovered at the network interface driver layer are passed by IP back up to the user process.

ERRORS

A socket operation may fail with one of the following errors returned:

EACCESS	when specifying an IP broadcast destination address if the caller is not the superuser;
EISCONN	when trying to establish a connection on a socket which already has one, or when trying to send a datagram with the destination address specified and the socket is already connected;
EMSGSIZE	when sending datagram that is too large for an interface, but is not allowed be fragmented (such as broadcasts);
ENETUNREACH	when trying to establish a connection or send a datagram, if there is no matching entry in the routing table, or if an ICMP "destination unreachable" message is received.
ENOTCONN	when trying to send a datagram, but no destination address is specified, and the socket hasn't been connected;
ENOBUFS	when the system runs out of memory for fragmentation buffers or other internal data structure;
EADDRNOTAVAIL	when an attempt is made to create a socket with a local address that matches no network interface, or when specifying an IP broadcast destination address and the network interface does not support broadcast;

The following errors may occur when setting or getting IP options:

EINVAL	An unknown socket option name was given.
EINVAL	The IP option field was improperly formed; an option field was shorter than the minimum value or longer than the option buffer provided.

SEE ALSO

connect(2), **getsockopt**(2), **read**(2V), **recv**(2), **send**(2), **write**(2V), **icmp**(4P), **inet**(4F) **routing**(4N), **tcp**(4P), **udp**(4P)

Postel, Jon, "*Internet Protocol - DARPA Internet Program Protocol Specification*," RFC 791, Network Information Center, SRI International, Menlo Park, Calif., September 1981. (Sun 800-1063-01)

BUGS

Raw sockets should receive ICMP error packets relating to the protocol; currently such packets are simply discarded.

Users of higher-level protocols such as TCP and UDP should be able to see received IP options.

NAME

kb – Sun keyboard STREAMS module

CONFIG

pseudo-device kb*number*

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sundev/vuid_event.h>
#include <sundev/kbio.h>
#include <sundev/kbd.h>

ioctl(fd, I_PUSH, "kb");
```

DESCRIPTION

The **kb** STREAMS module processes byte streams generated by Sun keyboards attached to a CPU serial or parallel port. Definitions for altering keyboard translation, and reading events from the keyboard, are in **<sundev/kbio.h>** and **<sundev/kbd.h>**. *number* specifies the maximum number of keyboards supported by the system.

kb recognizes which keys have been typed using a set of tables for each known type of keyboard. Each translation table is an array of 128 16-bit words (**unsigned shorts**). If an entry in the table is less than 0x100, it is treated as an ISO 8859/1 character. Higher values indicate special characters that invoke more complicated actions.

Keyboard Translation Mode

The keyboard can be in one of the following translation modes:

TR_NONE	Keyboard translation is turned off and up/down key codes are reported.
TR_ASCII	ISO 8859/1 codes are reported.
TR_EVENT	**firm_events** are reported (see *SunView Programmer's Guide*).
TR_UNTRANS_EVENT	**firm_events** containing unencoded keystation codes are reported for all input events within the window system.

Keyboard Translation-Table Entries

All instances of the **kb** module share seven translation tables used to convert raw keystation codes to event values. The tables are:

Unshifted	Used when a key is depressed and no shifts are in effect.
Shifted	Used when a key is depressed and a Shift key is being held down.
Caps Lock	Used when a key is depressed and Caps Lock is in effect.
Alt Graph	Used when a key is depressed and the Alt Graph key is being held down.
Num Lock	Used when a key is depressed and Num Lock is in effect.
Controlled	Used when a key is depressed and the Control key is being held down (regardless of whether a Shift key or the Alt Graph is being held down, or whether Caps Lock or Num Lock is in effect).
Key Up	Used when a key is released.

Each key on the keyboard has a "key station" code which is a number from 0 to 127. This number is used as an index into the translation table that is currently in effect. If the corresponding entry in that translation table is a value from 0 to 255, this value is treated as an ISO 8859/1 character, and that character is the result of the translation.

If the entry is a value above 255, it is a "special" entry. Special entry values are classified according to the value of the high-order bits. The high-order value for each class is defined as a constant, as shown in the list below. The value of the low-order bits, when added to this constant, distinguishes between keys within each class:

SHIFTKEYS 0x100 — A shift key. The value of the particular shift key is added to determine which shift mask to apply:

CAPSLOCK 0	"Caps Lock" key.
SHIFTLOCK 1	"Shift Lock" key.
LEFTSHIFT 2	Left-hand "Shift" key.
RIGHTSHIFT 3	Right-hand "Shift" key.
LEFTCTRL 4	Left-hand (or only) "Control" key.
RIGHTCTRL 5	Right-hand "Control" key.
ALTGRAPH 9	"Alt Graph" key.
ALT 10	"Alternate" key on the Sun-3 keyboard, or "Alt" key on the Sun-4 keyboard.
NUMLOCK 11	"Num Lock" key.

BUCKYBITS 0x200 — Used to toggle mode-key-up/down status without altering the value of an accompanying ISO 8859/1 character. The actual bit-position value, minus 7, is added.

METABIT 0	The "Meta" key was pressed along with the key. This is the only user-accessible bucky bit. It is ORed in as the 0x80 bit; since this bit is a legitimate bit in a character, the only way to distinguish between, for example, 0xA0 as META+0x20 and 0xA0 as an 8-bit character is to watch for "META key up" and "META key down" events and keep track of whether the META key was down.
SYSTEMBIT 1	The "System" key was pressed. This is a place holder to indicate which key is the system-abort key.

FUNNY 0x300 — Performs various functions depending on the value of the low 4 bits:

NOP 0x300	Does nothing.
OOPS 0x301	Exists, but is undefined.
HOLE 0x302	There is no key in this position on the keyboard, and the position-code should not be used.
NOSCROLL 0x303	Alternately sends CTRL-S and CTRL-Q characters.
CTRLS 0x304	Sends CTRL-S character and toggles NOSCROLL key.
CTRLQ 0x305	Sends CTRL-Q character and toggles NOSCROLL key.
RESET 0x306	Keyboard reset.
ERROR 0x307	The keyboard driver detected an internal error.
IDLE 0x308	The keyboard is idle (no keys down).
COMPOSE 0x309	This key is the COMPOSE key; the next two keys should comprise a two-character "COMPOSE key" sequence.

NONL 0x30A — Used only in the Num Lock table; indicates that this key is not affected by the Num Lock state, so that the translation table to use to translate this key should be the one that would have been used had Num Lock not been in effect.

0x30B — 0x30F — Reserved for nonparameterized functions.

FA_CLASS 0x400 — This key is a "floating accent" or "dead" key. When this key is pressed, the next key generates an event for an accented character; for example, "floating accent grave" followed by the "a" key generates an event with the ISO 8859/1 code for the "a with grave accent" character. The low-order bits indicate which accent; the codes for the individual "floating accents" are as follows:

FA_UMLAUT 0x400	umlaut
FA_CFLEX 0x401	circumflex
FA_TILDE 0x402	tilde
FA_CEDILLA 0x403	cedilla
FA_ACUTE 0x404	acute accent
FA_GRAVE 0x405	grave accent

STRING 0x500 — The low-order bits index a table of strings. When a key with a STRING entry is depressed, the characters in the null-terminated string for that key are sent, character by character. The maximum length is defined as:

KTAB_STRLEN 10

Individual string numbers are defined as:

HOMEARROW	0x00
UPARROW	0x01
DOWNARROW	0x02
LEFTARROW	0x03
RIGHTARROW	0x04

String numbers 0x05 — 0x0F are available for custom entries.

FUNCKEYS 0x600 — Function keys. The next-to-lowest 4 bits indicate the group of function keys:

LEFTFUNC 0x600
RIGHTFUNC 0x610
TOPFUNC 0x620
BOTTOMFUNC 0x630

The low 4 bits indicate the function key number within the group:

LF(*n*)	(LEFTFUNC+(*n*)-1)
RF(*n*)	(RIGHTFUNC+(*n*)-1)
TF(*n*)	(TOPFUNC+(*n*)-1)
BF(*n*)	(BOTTOMFUNC+(*n*)-1)

There are 64 keys reserved for function keys. The actual positions may not be on left/right/top/bottom of the keyboard, although they usually are.

PADKEYS 0x700

This key is a "numeric keypad key." These entries should appear only in the Num Lock translation table; when Num Lock is in effect, these events will be generated by pressing keys on the right-hand keypad. The low-order bits indicate which key; the codes for the individual keys are as follows:

PADEQUAL 0x700	"=" key
PADSLASH 0x701	"/" key
PADSTAR 0x702	"*" key
PADMINUS 0x703	"-" key
PADSEP 0x704	"," key
PAD7 0x705	"7" key
PAD8 0x706	"8" key
PAD9 0x707	"9" key
PADPLUS 0x708	"+" key
PAD4 0x709	"4" key
PAD5 0x70A	"5" key
PAD6 0x70B	"6" key
PAD1 0x70C	"1" key
PAD2 0x70D	"2" key
PAD3 0x70E	"3" key
PAD0 0x70F	"0" key
PADDOT 0x710	"." key
PADENTER 0x711	"Enter" key

In **TR_ASCII** mode, when a function key is pressed, the following escape sequence is sent:

ESC[0....9z

where ESC is a single escape character and "0...9" indicates the decimal representation of the function-key value. For example, function key **R1** sends the sequence:

ESC[208z

because the decimal value of RF(1) is 208. In **TR_EVENT** mode, if there is a VUID event code for the function key in question, an event with that event code is generated; otherwise, individual events for the characters of the escape sequence are generated.

Keyboard Compatibility Mode

kb is in "compatibility mode" when it starts up. In this mode, when the keyboard is in the **TR_EVENT** translation mode, ISO 8859/1 characters from the "upper half" of the character set (that is, characters with the 8th bit set) are presented as events with codes in the **ISO_FIRST** range (as defined in **<sundev/vuid_event.h>**). The event code is **ISO_FIRST** plus the character value. This is for backwards compatibility with older versions of the keyboard driver. If compatibility mode is turned off, ISO 8859/1 characters are presented as events with codes equal to the character code.

IOCTLS

The following **ioctl()** requests set and retrieve the current translation mode of a keyboard:

KIOCTRANS The argument is a pointer to an **int**. The translation mode is set to the value in the **int** pointed to by the argument.

KIOCGTRANS The argument is a pointer to an **int**. The current translation mode is stored in the **int** pointed to by the argument.

ioctl() requests for changing and retrieving entries from the keyboard translation table use the **kiockeymap** structure:

```
struct  kiockeymap {
        int     kio_tablemask;  /* Translation table (one of: 0, CAPSMASK,
                                   SHIFTMASK, CTRLMASK, UPMASK,
                                   ALTGRAPHMASK, NUMLOCKMASK) */
#define KIOCABORT1   -1         /* Special "mask": abort1 keystation */
#define KIOCABORT2   -2         /* Special "mask": abort2 keystation */
        u_char  kio_station;    /* Physical keyboard key station (0-127) */
        u_short kio_entry;      /* Translation table station's entry */
        char    kio_string[10]; /* Value for STRING entries (null terminated) */
};
```

KIOCSKEY The argument is a pointer to a **kiockeymap** structure. The translation table entry referred to by the values in that structure is changed.

kio_tablemask specifies which of the five translation tables contains the entry to be modified:

UPMASK 0x0080 "Key Up" translation table.

NUMLOCKMASK 0x0800 "Num Lock" translation table.

CTRLMASK 0x0030 "Controlled" translation table.

ALTGRAPHMASK 0x0200 "Alt Graph" translation table.

SHIFTMASK 0x000E "Shifted" translation table.

CAPSMASK 0x0001 "Caps Lock" translation table.

(No shift keys pressed or locked) "Unshifted" translation table.

kio_station specifies the keystation code for the entry to be modified. The value of **kio_entry** is stored in the entry in question. If **kio_entry** is between STRING and STRING+15, the string contained in **kio_string** is copied to the appropriate string table entry. This call may return **EINVAL** if there are invalid arguments.

There are a couple special values of **kio_tablemask** that affect the two step "break to the PROM monitor" sequence. The usual sequence is **SETUP–a** or **L1–a**. If **kio_tablemask** is **KIOCABORT1** then the value of **kio_station** is set to be the first keystation in the sequence. If **kio_tablemask** is **KIOCABORT2** then the value of **kio_station** is set to be the second keystation in the sequence.

KIOCGKEY The argument is a pointer to a **kiockeymap** structure. The current value of the keyboard translation table entry specified by **kio_tablemask** and **kio_station** is stored in the structure pointed to by the argument. This call may return **EINVAL** if there are invalid arguments.

KIOCTYPE The argument is a pointer to an **int**. A code indicating the type of the keyboard is stored in the **int** pointed to by the argument:

KB_KLUNK	Micro Switch 103SD32-2
KB_VT100	Keytronics VT100 compatible
KB_SUN2	Sun-2 keyboard
KB_SUN3	Sun-3 keyboard
KB_SUN4	Sun-4 keyboard
KB_ASCII	ASCII terminal masquerading as keyboard

–1 is stored in the **int** pointed to by the argument if the keyboard type is unknown.

KIOCLAYOUT The argument is a pointer to an **int**. On a Sun-4 keyboard, the layout code specified by the keyboard's DIP switches is stored in the **int** pointed to by the argument.

KIOCCMD The argument is a pointer to an **int**. The command specified by the value of the **int** pointed to by the argument is sent to the keyboard. The commands that can be sent are:

Commands to the Sun-2, Sun-3, and Sun-4 keyboard:

KBD_CMD_RESET	Reset keyboard as if power-up.
KBD_CMD_BELL	Turn on the bell.
KBD_CMD_NOBELL	Turn off the bell

Commands to the Sun-3 and Sun-4 keyboard:

KBD_CMD_CLICK	Turn on the click annunciator.
KBD_CMD_NOCLICK	Turn off the click annunciator.

Inappropriate commands for particular keyboard types are ignored. Since there is no reliable way to get the state of the bell or click (because we cannot query the keyboard, and also because a process could do writes to the appropriate serial driver — thus going around this **ioctl()** request) we do not provide an equivalent **ioctl()** to query its state.

KIOCSLED The argument is a pointer to an **char**. On the Sun-4 keyboard, the LEDs are set to the value specified in that **char**. The values for the four LEDs are:

LED_CAPS_LOCK	"Caps Lock" light.
LED_COMPOSE	"Compose" light.
LED_SCROLL_LOCK	"Scroll Lock" light.
LED_NUM_LOCK	"Num Lock" light.

KIOCGLED The argument is a pointer to a **char**. The current state of the LEDs is stored in the **char** pointed to by the argument.

KIOCSCOMPAT The argument is a pointer to an **int**. "Compatibility mode" is turned on if the **int** has a value of 1, and is turned off if the **int** has a value of 0.

KIOCGCOMPAT The argument is a pointer to an **int**. The current state of "compatibility mode" is stored in the **int** pointed to by the argument.

KIOCGDIRECT These **ioctl()** requests are supported for compatibility with the system keyboard device **/dev/kbd**. **KIOCSDIRECT** has no effect, and **KIOCGDIRECT** always returns 1.

SEE ALSO

click(1), **loadkeys**(1), **kbd**(4S), **termio**(4), **win**(4S), **keytables**(5)

SunView Programmer's Guide (describes **firm_event** format)

NAME

kbd – Sun keyboard

CONFIG

None; included in standard system.

DESCRIPTION

The **kbd** device provides access to the Sun Workstation keyboard. When opened, it provides access to the standard keyboard device for the workstation (attached either to a CPU serial or parallel port). It is a multiplexing driver; a stream referring to the standard keyboard device, with the **kb**(4M) and **ttcompat**(4M) STREAMS modules pushed on top of that device, is linked below it. Normally, this device passes input to the "workstation console" driver, which is linked above a special minor device of **kbd**, so that keystrokes appear as input on **/dev/console**; the **KIOCSDIRECT ioctl** must be used to direct input towards or away from the **/dev/kbd** device.

IOCTLS

KIOCSDIRECT The argument is a pointer to an **int**. If the value in the **int** pointed to by the argument is 1, subsequent keystrokes typed on the system keyboard will sent to **/dev/kbd**; if it is 0, subsequent keystrokes will be sent to the "workstation console" device. When the last process that has **/dev/kbd** open closes it, if keystrokes had been sent to **/dev/kbd** they are redirected back to the "workstation console" device.

KIOCGDIRECT The argument is a pointer to an **int**. If keystrokes are currently being sent to **/dev/kbd**, 1 is stored in the **int** pointed to by the argument; if keystrokes are currently being sent to the "workstation console" device, 0 is stored there.

FILES

/dev/kbd

SEE ALSO

console(4S), **kb**(4M), **ttcompat**(4M), **win**(4S), zs(4S)

NAME

ldterm – standard terminal STREAMS module

CONFIG

None; included by default.

SYNOPSIS

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

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

DESCRIPTION

ldterm is a STREAMS module that provides most of the **termio**(4) terminal interface. This module does not perform the low-level device control functions specified by flags in the **c_cflag** word of the **termios** structure or by the **IGNBRK**, **IGNPAR**, **PARMRK**, or **INPCK** flags in the **c_iflag** word of the **termios** structure; those functions must be performed by the driver or by modules pushed below the **ldterm** module. All other **termio** functions are performed by **ldterm**; some of them, however, require the cooperation of the driver or modules pushed below **ldterm**, and may not be performed in some cases. These include the **IXOFF** flag in the **c_iflag** word and the delays specified in the **c_oflag** word.

Read-side Behavior

Various types of STREAMS messages are processed as follows:

M_BREAK When this message is received, either an interrupt signal is generated, or the message is treated as if it were an **M_DATA** message containing a single ASCII NUL character, depending on the state of the **BRKINT** flag.

M_DATA These messages are normally processed using the standard **termio** input processing. If the **ICANON** flag is set, a single input record ("line") is accumulated in an internal buffer, and sent upstream when a line-terminating character is received. If the **ICANON** flag is not set, other input processing is performed and the processed data is passed upstream.

If output is to be stopped or started as a result of the arrival of characters, **M_STOP** and **M_START** messages are sent downstream, respectively. If the **IXOFF** flag is set, and input is to be stopped or started as a result of flow-control considerations, **M_STOPI** and **M_STARTI** messages are sent downstream, respectively.

M_DATA messages are sent downstream, as necessary, to perform echoing.

If a signal is to be generated, a **M_FLUSH** message with a flag byte of **FLUSHR** is placed on the read queue, and if the signal is also to flush output a **M_FLUSH** message with a flag byte of **FLUSHW** is sent downstream.

M_CTL If the first byte of the message is **MC_NOCANON**, the input processing normally performed on **M_DATA** messages is disabled, and those messages are passed upstream unmodified; this is for the use of modules or drivers that perform their own input processing, such as a pseudo-terminal in **TIOCREMOTE** mode connected to a program that performs this processing. If the first byte of the message is **MC_DOCANON**, the input processing is enabled. Otherwise, the message is ignored; in any case, the message is passed upstream.

M_FLUSH The read queue of the module is flushed of all its data messages, and all data in the record being accumulated is also flushed. The message is passed upstream.

M_HANGUP Data is flushed as it is for a **M_FLUSH** message, and **M_FLUSH** messages with a flag byte of **FLUSHRW** are sent upstream and downstream. Then an **M_PCSIG** message is sent upstream with a signal of **SIGCONT**, followed by the **M_HANGUP** message.

M_IOCACK The data contained within the message, which is to be returned to the process, is augmented if necessary, and the message is passed upstream.

All other messages are passed upstream unchanged.

Write-side behavior

Various types of STREAMS messages are processed as follows:

M_FLUSH The write queue of the module is flushed of all its data messages, and the message is passed downstream.

M_IOCTL The function to be performed for this **ioctl()** request by the **ldterm** module is performed, and the message is passed downstream in most cases. The **TCFLSH** and **TCXONC** **ioctl()** requests can be performed entirely in this module, so the reply is sent upstream and the message is not passed downstream.

M_DATA If the **OPOST** flag is set, or both the **XCASE** and **ICANON** flags are set, output processing is performed and the processed message is passed downstream, along with any **M_DELAY** messages generated. Otherwise, the message is passed downstream without change.

All other messages are passed downstream unchanged.

IOCTLS

The following **ioctl()** requests are processed by the **ldterm** module. All others are passed downstream.

TCGETS
TCGETA The message is passed downstream; if an acknowledgment is seen, the data provided by the driver and modules downstream is augmented and the acknowledgement is passed upstream.

TCSETS
TCSETSW
TCSETSF
TCSETA
TCSETAW
TCSETAF The parameters that control the behavior of the **ldterm** module are changed. If a mode change requires options at the stream head to be changed, a **M_SETOPT** 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 or byte-stream mode, respectively. If it is turned on, the **vmin** and **vtime** values at the stream head are set to 1 and 0, respectively; if it is turned on, they are set to the values specified by the **ioctl()** request. The **vmin** and **vtime** values are also set if **ICANON** is off and the values are changed by the **ioctl()** request. If the **TOSTOP** flag is turned on or off, the **tostop** mode at the stream head is turned on or off, respectively.

TCFLSH If the argument is 0, an **M_FLUSH** message with a flag byte of **FLUSHR** is sent downstream and placed on the read queue. If the argument is 1, the write queue is flushed of all its data messages and a **M_FLUSH** message with a flag byte of **FLUSHW** is sent upstream and downstream. If the argument is 2, the write queue is flushed of all its data messages and a **M_FLUSH** message with a flag byte of **FLUSHRW** is sent downstream and placed on the read queue.

TCXONC If the argument is 0, and output is not already stopped, an **M_STOP** message is sent downstream. If the argument is 1, and 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.

SEE ALSO

console(4S), **mcp**(4S), **mti**(4S), **pty**(4), **termio**(4), **ttcompat**(4M), **zs**(4S)

NAME

ie – Intel 10 Mb/s Ethernet interface

CONFIG — SUN-4 SYSTEM

device ie0 at obio ? csr 0xf6000000 priority 3
device ie1 at vme24d16 ? csr 0xe88000 priority 3 vector ieintr 0x75
device ie2 at vme24d16 ? csr 0x31ff02 priority 3 vector ieintr 0x76
device ie3 at vme24d16 ? csr 0x35ff02 priority 3 vector ieintr 0x77

CONFIG — SUN-3x SYSTEM

device ie0 at obio ? csr 0x65000000 priority 3
device ie1 at vme24d16 ? csr 0xe88000 priority 3 vector ieintr 0x75

CONFIG — SUN-3 SYSTEM

device ie0 at obio ? csr 0xc0000 priority 3
device ie1 at vme24d16 ? csr 0xe88000 priority 3 vector ieintr 0x75
device ie2 at vme24d32 ? csr 0x31ff02 priority 3 vector ieintr 0x76
device ie3 at vme24d32 ? csr 0x35ff02 priority 3 vector ieintr 0x77

CONFIG — SUN-3E SYSTEM

device ie0 at vme24d16 ? csr 0x31ff02 priority 3 vector ieintr 0x74

CONFIG — SUN386i SYSTEM

device ie0 at obmem ? csr 0xD0000000 irq 21 priority 3

DESCRIPTION

The **ie** interface provides access to a 10 Mb/s Ethernet network through a controller using the Intel 82586 LAN Coprocessor chip. For a general description of network interfaces see **if**(4N).

ie0 specifies a CPU-board-resident interface, except on a Sun-3E where **ie0** is the Sun-3/E Ethernet expansion board. **ie1** specifies a Multibus Intel Ethernet interface for use with a VME adapter. **ie2** and **ie3** specify SunNet Ethernet/VME Controllers, also known as a Sun-3/E Ethernet expansion boards.

SEE ALSO

if(4N), **le**(4S)

DIAGNOSTICS

There are too many driver messages to list them all individually here. Some of the more common messages and their meanings follow.

ie%d: Ethernet jammed
Network activity has become so intense that sixteen successive transmission attempts failed, and the 82586 gave up on the current packet. Another possible cause of this message is a noise source somewhere in the network, such as a loose transceiver connection.

ie%d: no carrier
The 82586 has lost input to its carrier detect pin while trying to transmit a packet, causing the packet to be dropped. Possible causes include an open circuit somewhere in the network and noise on the carrier detect line from the transceiver.

ie%d: lost interrupt: resetting
The driver and 82586 chip have lost synchronization with each other. The driver recovers by resetting itself and the chip.

ie%d: iebark reset
The 82586 failed to complete a watchdog timeout command in the allotted time. The driver recovers by resetting itself and the chip.

ie%d: WARNING: requeuing

The driver has run out of resources while getting a packet ready to transmit. The packet is put back on the output queue for retransmission after more resources become available.

ie%d: panic: scb overwritten

The driver has discovered that memory that should remain unchanged after initialization has become corrupted. This error usually is a symptom of a bad 82586 chip.

ie%d: giant packet

Provided that all stations on the Ethernet are operating according to the Ethernet specification, this error "should never happen," since the driver allocates its receive buffers to be large enough to hold packets of the largest permitted size. The most likely cause of this message is that some other station on the net is transmitting packets whose lengths exceed the maximum permitted for Ethernet.

NAME

lo – software loopback network interface

SYNOPSIS

pseudo-device loop

DESCRIPTION

The **loop** device is a software loopback network interface; see **if**(4N) for a general description of network interfaces.

The **loop** interface is used for performance analysis and software testing, and to provide guaranteed access to Internet protocols on machines with no local network interfaces. A typical application is the **comsat**(8C) server which accepts notification of mail delivery through a particular port on the loopback interface.

By default, the loopback interface is accessible at Internet address 127.0.0.1 (non-standard); this address may be changed with the **SIOCSIFADDR** ioctl.

SEE ALSO

if(4N), **inet**(4F), **comsat**(8C)

DIAGNOSTICS

lo%d: can't handle af%d

The interface was handed a message with addresses formatted in an unsuitable address family; the packet was dropped.

BUGS

It should handle all address and protocol families. An approved network address should be reserved for this interface.

NAME

lofs – loopback virtual file system

CONFIG

options LOFS

SYNOPSIS

#include <sys/mount.h>
mount(MOUNT_LOFS, virtual, flags, dir);

DESCRIPTION

The loopback file system device allows new, virtual file systems to be created, which provide access to existing files using alternate pathnames. Once the virtual file system is created, other file systems can be mounted within it without affecting the original file system. File systems that are subsequently mounted onto the original file system, however, *are* visible to the virtual file system, unless or until the corresponding mount point in the virtual file system is covered by a file system mounted there.

virtual is the mount point for the virtual file system. *dir* is the pathname of the existing file system. *flags* is either 0 or M_RDONLY. The M_RDONLY flag forces all accesses in the new name space to be read-only; without it, accesses are the same as for the underlying file system. All other **mount**(2V) flags are preserved from the underlying file systems.

A loopback mount of '/' onto **/tmp/newroot** allows the entire file system hierarchy to appear as if it were duplicated under **/tmp/newroot**, including any file systems mounted from remote NFS servers. All files would then be accessible either from a pathname relative to '/', or from a pathname relative to **/tmp/newroot** until such time as a file system is mounted in **/tmp/newroot**, or any of its subdirectories.

Loopback mounts of '/' can be performed in conjunction with the **chroot**(2) system call, to provide a complete virtual file system to a process or family of processes.

Recursive traversal of loopback mount points is not allowed; after the loopback mount of **/tmp/newroot**, the file **/tmp/newroot/tmp/newroot** does not contain yet another file system hierarchy; rather, it appears just as **/tmp/newroot** did before the loopback mount was performed (say, as an empty directory).

The standard RC files perform first **4.2** mounts, then **nfs** mounts, during booting. On Sun386i systems, **lo** (loopback) mounts are performed just after **4.2** mounts. **/etc/fstab** files depending on alternate mount orders at boot time will fail to work as expected. Manual modification of **/etc/rc.local** will be needed to make such mount orders work.

WARNINGS

Loopback mounts must be used with care; the potential for confusing users and applications is enormous. A loopback mount entry in **/etc/fstab** must be placed after the mount points of both directories it depends on. This is most easily accomplished by making the loopback mount entry the last in **/etc/fstab**, though see **mount**(8) for further warnings.

SEE ALSO

chroot(2), **mount**(2V), **fstab**(5), **mount**(8)

BUGS

Because only directories can be mounted or mounted on, the structure of a virtual file system can only be modified at directories.

NAME

mcp, alm – Sun MCP Multiprotocol Communications Processor/ALM-2 Asynchronous Line Multiplexer

CONFIG — SUN-3, SUN-4 SYSTEMS

MCP

device mcp0 at vme32d32 ? csr 0x1000000 flags 0x1ffff priority 4 vector mcpintr 0x8b
device mcp1 at vme32d32 ? csr 0x1010000 flags 0x1ffff priority 4 vector mcpintr 0x8a
device mcp2 at vme32d32 ? csr 0x1020000 flags 0x1ffff priority 4 vector mcpintr 0x89
device mcp3 at vme32d32 ? csr 0x1030000 flags 0x1ffff priority 4 vector mcpintr 0x88

ALM-2

pseudo-device mcpa64

CONFIG — SUN-3x SYSTEMS

MCP

device mcp0 at vme32d32 ? csr 0x1000000 flags 0x1ffff priority 4 vector mcpintr 0x8b
device mcp1 at vme32d32 ? csr 0x1010000 flags 0x1ffff priority 4 vector mcpintr 0x8a
device mcp2 at vme32d32 ? csr 0x1020000 flags 0x1ffff priority 4 vector mcpintr 0x89
device mcp3 at vme32d32 ? csr 0x1030000 flags 0x1ffff priority 4 vector mcpintr 0x88
device mcp4 at vme32d32 ? csr 0x1040000 flags 0x1ffff priority 4 vector mcpintr 0xa0
device mcp5 at vme32d32 ? csr 0x1050000 flags 0x1ffff priority 4 vector mcpintr 0xa1
device mcp6 at vme32d32 ? csr 0x1060000 flags 0x1ffff priority 4 vector mcpintr 0xa2
device mcp7 at vme32d32 ? csr 0x1070000 flags 0x1ffff priority 4 vector mcpintr 0xa3

ALM-2

pseudo-device mcpa64

SYNOPSIS

#include <fcntl.h>
#include <sys/termios.h>
open("/dev/tty*xy*", mode);
open("/dev/ttyd*n*", mode);
open("/dev/cua*n*", mode);

DESCRIPTION (MCP)

The Sun MCP (Multiprotocol Communications Processor) supports up to four synchronous serial lines in conjunction with SunLink™ Multiple Communication Protocol products.

DESCRIPTION (ALM-2)

The Sun ALM-2 Asynchronous Line Multiplexer provides 16 asynchronous serial communication lines with modem control and one Centronics-compatible parallel printer port.

Each port supports those **termio**(4) device control functions specified by flags in the **c_cflag** word of the **termios** structure and by the **IGNBRK**, **IGNPAR**, **PARMRK**, or **INPCK** flags in the **c_iflag** word of the **termios** structure are performed by the **mcp** driver. All other **termio**(4) functions must be performed by STREAMS modules pushed atop the driver; when a device is opened, the **ldterm**(4M) and **ttcompat**(4M) STREAMS modules are automatically pushed on top of the stream, providing the standard **termio**(4) interface.

Bit *i* of **flags** may be specified to say that a line is not properly connected, and that the line *i* should be treated as hard-wired with carrier always present. Thus specifying **flags 0x0004** in the specification of **mcp0** would treat line **/dev/ttyh2** in this way.

Minor device numbers in the range 0 – 63 correspond directly to the normal tty lines and are named **/dev/tty***XY*, where *X* represents the physical board as one of the characters **h**, **i**, **j**, or **k**, and *Y* is the line number on the board as a single hexadecimal digit. (Thus the first line on the first board is **/dev/ttyh0**, and the sixteenth line on the third board is **/dev/ttyjf**.)

To allow a single tty line to be connected to a modem and used for both incoming and outgoing calls, a special feature, controlled by the minor device number, has been added. Minor device numbers in the range 128 – 191 correspond to the same physical lines as those above (that is, the same line as the minor device number minus 128).

A dial-in line has a minor device in the range 0 – 63 and is conventionally renamed **/dev/ttyd***n*, where *n* is a number indicating which dial-in line it is (so that **/dev/ttyd0** is the first dial-in line), and the dial-out line corresponding to that dial-in line has a minor device number 128 greater than the minor device number of the dial-in line and is conventionally named **/dev/cua***n*, where *n* is the number of the dial-in line.

The **/dev/cua***n* lines are special in that they can be opened even when there is no carrier on the line. Once a **/dev/cua***n* line is opened, the corresponding tty line cannot be opened until the **/dev/cua***n* line is closed; a blocking open will wait until the **/dev/cua***n* line is closed (which will drop Data Terminal Ready, after which Carrier Detect will usually drop as well) and carrier is detected again, and a non-blocking open will return an error. Also, if the **/dev/ttyd***n* line has been opened successfully (usually only when carrier is recognized on the modem) the corresponding **/dev/cua***n* line cannot be opened. This allows a modem to be attached to e.g. **/dev/ttyd0** (renamed from **/dev/ttyh0**) and used for dialin (by enabling the line for login in **/etc/ttytab**) and also used for dialout (by **tip**(1C) or **uucp**(1C)) as **/dev/cua0** when no one is logged in on the line. Note: the bit in the flags word in the configuration file (see above) must be zero for this line, which enables hardware carrier detection.

IOCTLS

The standard set of **termio ioctl()** calls are supported by the ALM-2.

If the CRTSCTS flag in the **c_cflag** is set, output will be generated only if CTS is high; if CTS is low, output will be frozen. If the CRTSCTS flag is clear, the state of CTS has no effect. Breaks can be generated by the **TCSBRK**, **TIOCSBRK**, and **TIOCCBRK ioctl()** calls. The modem control lines **TIOCM_CAR**, **TIOCM_CTS**, **TIOCM_RTS**, and **TIOCM_DTR** are provided.

The input and output line speeds may be set to any of the speeds supported by **termio**. The speeds cannot be set independently; when the output speed is set, the input speed is set to the same speed.

ERRORS

An **open()** on a **/dev/tty*** or a **/dev/cu*** device will fail if:

ENXIO	The unit being opened does not exist.
EBUSY	The dial-out device is being opened and the dial-in device is already open, or the dial-in device is being opened with a no-delay open and the dial-out device is already open.
EBUSY	The unit has been marked as exclusive-use by another process with a **TIOCEXCL ioctl()** call.
EINTR	The open was interrupted by the delivery of a signal.

DESCRIPTION (PRINTER PORT)

The printer port is Centronics-compatible and is suitable for most common parallel printers. Devices attached to this interface are normally handled by the line printer spooling system, and should not be accessed directly by the user.

Minor device numbers in the range 64 – 67 access the printer port, and the recommended naming is **/dev/mcpp[0-3]**.

IOCTLS

Various control flags and status bits may be fetched and set on an MCP printer port. The following flags and status bits are supported; they are defined in **sundev/mcpcmd.h**:

MCPRIGNSLCT	0x02	set if interface ignoring SLCT– on open
MCPRDIAG	0x04	set if printer is in self-test mode
MCPRVMEINT	0x08	set if VME bus interrupts enabled
MCPRINTPE	0x10	print message when out of paper
MCPRINTSLCT	0x20	print message when printer offline

MCPRPE	0x40	set if device ready, cleared if device out of paper
MCPRSLCT	0x80	set if device online (Centronics SLCT asserted)

The flags **MCPRINTSLCT, MCPRINTPE,** and **MCPRDIAG** may be changed; the other bits are status bits and may not be changed.

The **ioctl()** calls supported by MCP printer ports are listed below.

MCPIOGPR The argument is a pointer to an **unsigned char**. The printer flags and status bits are stored in the **unsigned char** pointed to by the argument.

MCPIOSPR The argument is a pointer to an **unsigned char**. The printer flags are set from the **unsigned char** pointed to by the argument.

ERRORS

Normally, the interface only reports the status of the device when attempting an **open**(2V) call. An **open()** on a **/dev/mcpp*** device will fail if:

ENXIO The unit being opened does not exist.

EIO The device is offline or out of paper.

Bit 17 of the configuration **flags** may be specified to say that the interface should ignore Centronics SLCT– and RDY/PE– when attempting to open the device, but this is normally useful only for configuration and troubleshooting: if the SLCT– and RDY lines are not asserted during an actual data transfer (as with a **write**(2V) call), no data is transferred.

FILES

/dev/mcpp[0-3]	parallel printer port
/dev/tty[h-k][0-9a-f]	hardwired tty lines
/dev/ttyd[0-9a-f]	dialin tty lines
/dev/cua[0-9a-f]	dialout tty lines

SEE ALSO

tip(1C), **uucp**(1C), **mti**(4S), **termio**(4), **ldterm**(4M), **ttcompat**(4M), **zs**(4S), **ttysoftcar**(8)

DIAGNOSTICS

Most of these diagnostics "should never happen;" their occurrence usually indicates problems elsewhere in the system as well.

mcpa*n*: silo overflow.
More than *n* characters (*n* very large) have been received by the **mcp** hardware without being read by the software.

port *n* supports RS449 interface**
Probably an incorrect jumper configuration. Consult the hardware manual.

mcp port *n* receive buffer error
The **mcp** encountered an error concerning the synchronous receive buffer.

Printer on mcpp*n* is out of paper
Printer on mcpp*n* paper ok
Printer on mcpp*n* is offline
Printer on mcpp*n* online
Assorted printer diagnostics, if enabled as discussed above.

BUGS

Note: pin 4 is used for hardware flow control on ALM–2 ports 0 through 3. These two pins should *not* be tied together on the ALM end.

NAME

mem, kmem, zero, vme16d16, vme24d16, vme32d16, vme16d32, vme24d32, vme32d32, eeprom, atbus, sbus – main memory and bus I/O space

CONFIG

None; included with standard system.

DESCRIPTION

These devices are special files that map memory and bus I/O space. They may be read, written, seeked and (except for **kmem**) memory-mapped. See **read**(2V), **write**(2V), **mmap**(2), and **directory**(3V).

All Systems

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

kmem is a special file that is an image of the kernel virtual memory of the system.

zero is a special file which is a source of private zero pages.

eeprom is a special file that is an image of the EEPROM or NVRAM.

Sun-3 and Sun-4 Systems VMEbus

vme16d16 (also known as **vme16**) is a special file that is an image of VMEbus 16-bit addresses with 16-bit data. **vme16** address space extends from 0 to 64K.

vme24d16 (also known as **vme24**) is a special file that is an image of VMEbus 24-bit addresses with 16-bit data. **vme24** address space extends from 0 to 16 Megabytes. The VME 16-bit address space overlaps the top 64K of the 24-bit address space.

vme32d16 is a special file that is an image of VMEbus 32-bit addresses with 16-bit data.

vme16d32 is a special file that is an image of VMEbus 16-bit addresses with 32-bit data.

vme24d32 is a special file that is an image of VMEbus 24-bit addresses with 32-bit data.

vme32d32 (also known as **vme32**) is a special file that is an image of VMEbus 32-bit addresses with 32-bit data. **vme32** address space extends from 0 to 4 Gigabytes. The VME 24-bit address space overlaps the top 16 Megabytes of the 32-bit address space.

SPARCstation 1 Systems

The **sbus** is represented by a series of entries each of which is an image of a single **sbus** slot. The entries are named **sbus***n*, where *n* is the slot number in hexadecimal. The number of **sbus** slots and the address range within each slot may vary between implementations.

Sun386i Systems

atbus is a special file that is an image of the AT bus space. It extends from 0 to 16 Megabytes.

FILES

/dev/mem
/dev/kmem
/dev/zero
/dev/vme16d16
/dev/vme16
/dev/vme24d16
/dev/vme24
/dev/vme32d16
/dev/vme16d32
/dev/vme24d32
/dev/vme32d32
/dev/vme32
/dev/eeprom
/dev/atbus
/dev/sbus[0-3]

SEE ALSO

mmap(2), **read**(2V), **write**(2V), **directory**(3V)

NAME

mouse – Sun mouse

CONFIG

None; included in standard system.

DESCRIPTION

The **mouse** indirect device provides access to the Sun Workstation mouse. When opened, it redirects operations to the standard mouse device for the workstation (attached either to a CPU serial or parallel port), and pushes the **ms**(4M) and **ttcompat**(4M) STREAMS modules on top of that device.

FILES

/dev/mouse

SEE ALSO

ms(4M), **ttcompat**(4M), **win**(4S), **zs**(4S)

NAME

ms – Sun mouse STREAMS module

CONFIG

pseudo-device*ms***n**

SYNOPSIS

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sundev/vuid_event.h>
#include <sundev/msio.h>
ioctl(fd, I_PUSH, "ms");
```

DESCRIPTION

The **ms** STREAMS module processes byte streams generated by mice attached to a CPU serial or parallel port. When this module is pushed onto a stream, it sends a **TCSETSF** ioctl downstream, setting the baud rate to 1200 baud and the character size to 8 bits, and enabling the receiver. All other flag words are cleared. It assumes only that the **termios**(3V) functions provided by the **zs**(4S) driver are supported; no other functions need be supported.

The mouse is expected to generate a stream of bytes encoding mouse motions and changes in the state of the buttons.

Each mouse sample in the byte stream consists of three bytes: the first byte gives the button state with value 0x87|~*but*, where *but* is the low three bits giving the mouse buttons, where a 0 (zero) bit means that a button is pressed, and a 1 (one) bit means a button is not pressed. Thus if the left button is down the value of this sample is 0x83, while if the right button is down the byte is 0x86.

The next two bytes of each sample give the *x* and *y* deltas of this sample as signed bytes. The mouse uses a lower-left coordinate system, so moves to the right on the screen yield positive *x* values and moves down the screen yield negative *y* values.

The beginning of a sample is identifiable because the delta's are constrained to not have values in the range 0x80-0x87.

A stream with **ms** pushed onto it can be used as a device that emits *firm_events* as specified by the protocol of a *Virtual User Input Device*. It understands **VUIDSFORMAT, VUIDGFORMAT, VUIDSADDR** and **VUIDGADDR** ioctls (see reference below).

IOCTLS

ms responds to the following *ioctl*s, as defined in **<sundev/msio.h>** and **<sundev/vuid_event.h>**. All other *ioctl*s are passed downstream. As **ms** sets the parameters of the serial port when it is opened, no **termios**(3V) *ioctl*s should be performed on a stream with **ms** on it, as **ms** expects the device parameters to remain as it set them.

The **MSIOGETPARMS** and **MSIOSETPARMS** calls use a structure of type **Ms_parms**, which is a structure defined in **<sundev/msio.h>**:

```
typedef struct {
int         jitter_thresh;
int         speed_law;
int         speed_limit;
}           Ms_parms;
```

jitter_thresh is the "jitter threshold" of the mouse. Motions of fewer than *jitter_thresh* units along both axes that occur in less than 1/12 second are treated as "jitter" and ignored. Thus, if the mouse moves fewer than *jitter_thresh units* and then moves back to its original position in less than 1/12 of a second, the motion is considered to be "noise" and ignored. If it moves fewer than *jitter_thresh* units and continues to move so that it has not returned to its original position after 1/12 of a second, the motion is considered to be real and is reported.

speed_law indicates whether extremely large motions are to be ignored. If it is 1, a "speed limit" is applied to mouse motions; motions along either axis of more than *speed_limit* units are discarded.

Note: these parameters are global; if they are set for any mouse on a workstation, they apply to any other mice attached to that workstation as well.

VUIDSFORMAT
VUIDGFORMAT
VUIDSADDR
VUIDGADDR These are standard *Virtual User Input Device ioctl*s. See *SunView System Programmer's Guide* for a description of their operation.

MSIOGETPARMS The argument is a pointer to a **Ms_parms**. The current mouse parameters are stored in that structure.

MSIOSETPARMS The argument is a pointer to a **ms_parms**. The current mouse parameters are set from the values in that structure.

SEE ALSO

mouse(4S), **termios**(3V), **win**(4S), **zs**(4S)

SunView System Programmer's Guide

NAME

mti – Systech MTI-800/1600 multi-terminal interface

CONFIG — SUN-3, SUN-3x, SUN-4 SYSTEMS

device mti0 at vme16d16 ? csr 0x620 flags 0xffff priority 4 vector mtiintr 0x88
device mti1 at vme16d16 ? csr 0x640 flags 0xffff priority 4 vector mtiintr 0x89
device mti2 at vme16d16 ? csr 0x660 flags 0xffff priority 4 vector mtiintr 0x8a
device mti3 at vme16d16 ? csr 0x680 flags 0xffff priority 4 vector mtiintr 0x8b

SYNOPSIS

#include <fcntl.h>
#include <sys/termios.h>
open("/dev/tty*xy*", mode);
open("/dev/ttyd*n*", mode);
open("/dev/cua*n*", mode);

DESCRIPTION

The Systech MTI card provides 8 (MTI-800) or 16 (MTI-1600) serial communication lines with modem control. Each port supports those **termio**(4) device control functions specified by flags in the **c_cflag** word of the **termios** structure and by the **IGNBRK**, **IGNPAR**, **PARMRK**, or **INPCK** flags in the **c_iflag** word of the **termios** structure are performed by the **mti** driver. All other **termio**(4) functions must be performed by STREAMS modules pushed on top of the driver; when a device is opened, the **ldterm**(4M) and **ttcompat**(4M) STREAMS modules are automatically pushed on top of the stream, providing the standard **termio**(4) interface.

Bit *i* of **flags** may be specified to say that a line is not properly connected, and that the line *i* should be treated as hard-wired with carrier always present. Thus specifying **flags 0x0004** in the specification of **mti0** would treat line **/dev/tty02** in this way.

Minor device numbers in the range 0 – 63 correspond directly to the normal tty lines and are named **/dev/tty***XY*, where *X* is the physical board number (0 – 3), and *Y* is the line number on the board as a single hexadecimal digit. Thus the first line on the first board is **/dev/tty00**, and the sixteenth line on the third board is **/dev/tty2f**.

To allow a single tty line to be connected to a modem and used for both incoming and outgoing calls, a special feature, controlled by the minor device number, has been added. Minor device numbers in the range 128 – 191 correspond to the same physical lines as those above (that is, the same line as the minor device number minus 128).

A dial-in line has a minor device in the range 0 – 63 and is conventionally renamed **/dev/ttyd***n*, where *n* is a number indicating which dial-in line it is (so that **/dev/ttyd0** is the first dial-in line), and the dial-out line corresponding to that dial-in line has a minor device number 128 greater than the minor device number of the dial-in line and is conventionally named **/dev/cua***n*, where *n* is the number of the dial-in line.

The **/dev/cua***n* lines are special in that they can be opened even when there is no carrier on the line. Once a **/dev/cua***n* line is opened, the corresponding tty line can not be opened until the **/dev/cua***n* line is closed; a blocking open will wait until the **/dev/cua***n* line is closed (which will drop Data Terminal Ready, after which Carrier Detect will usually drop as well) and carrier is detected again, and a non-blocking open will return an error. Also, if the **/dev/ttyd***n* line has been opened successfully (usually only when carrier is recognized on the modem) the corresponding **/dev/cua***n* line can not be opened. This allows a modem to be attached to for example, **/dev/ttyd0** (renamed from **/dev/tty00**) and used for dial-in (by enabling the line for login in **/etc/ttytab**) and also used for dial-out (by **tip**(1C) or **uucp**(1C)) as **/dev/cua0** when no one is logged in on the line. Note: the bit in the **flags** word in the configuration file (see above) must be zero for this line, which enables hardware carrier detection.

WIRING

The Systech requires the CTS modem control signal to operate. If the device does not supply CTS then RTS should be jumpered to CTS at the distribution panel (short pins 4 to 5). Also, the CD (carrier detect) line does not work properly. When connecting a modem, the modem's CD line should be wired to DSR, which the software will treat as carrier detect.

IOCTLS

The standard set of **termio ioctl()** calls are supported by **mti**.

The state of the CRTSCTS flag in the **c_cflag** word has no effect; no output will be generated unless CTS is high. Breaks can be generated by the **TCSBRK**, **TIOCSBRK**, and **TIOCCBRK ioctl()** calls. The modem control lines **TIOCM_CAR**, **TIOCM_CTS**, **TIOCM_RTS**, and **TIOCM_DTR** are provided; however, as described above, the DSR line is treated as CD and the CD line is ignored.

The input and output line speeds may be set to any of the speeds supported by **termio**. The speeds cannot be set independently; when the output speed is set, the input speed is set to the same speed. The baud rates **B200** and **B38400** are not supported by the hardware; **B200** selects 2000 baud, and **B38400** selects 7200 baud.

ERRORS

An **open()** will fail if:

ENXIO	The unit being opened does not exist.
EBUSY	The dial-out device is being opened and the dial-in device is already open, or the dial-in device is being opened with a no-delay open and the dial-out device is already open.
EBUSY	The unit has been marked as exclusive-use by another process with a **TIOCEXCL ioctl()** call.
EINTR	The open was interrupted by the delivery of a signal.

FILES

/dev/tty[0-3][0-9a-f]	hardwired tty lines
/dev/ttyd[0-9a-f]	dial-in tty lines
/dev/cua[0-9a-f]	dial-out tty lines

SEE ALSO

tip(1C), **uucp**(1C), **mcp**(4S), **termio**(4), **ldterm**(4M), **ttcompat**(4M), **zs**(4S), **ttysoftcar**(8)

DIAGNOSTICS

Most of these diagnostics "should never happen" and their occurrence usually indicates problems elsewhere in the system.

mti*n*, *n*: silo overflow.
More than 512 characters have been received by the **mti** hardware without being read by the software. Extremely unlikely to occur.

mti*n*: read error code <*n*>. Probable hardware fault
The **mti** returned the indicated error code. See the MTI manual.

mti*n*: DMA output error.
The **mti** encountered an error while trying to do DMA output.

mti*n*: impossible response *n*.
The **mti** returned an error it could not understand.

NAME

mtio – general magnetic tape interface

SYNOPSIS

#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/mtio.h>

DESCRIPTION

1/2", 1/4" and 8 mm magnetic tape drives all share the same general character device interface.

There are two types of tape records: data records and end-of-file (EOF) records. EOF records are also known as tape marks and file marks. A record is separated by interrecord (or tape) gaps on a tape.

End-of-recorded-media (EOM) is indicated by two EOF marks on 1/2" tape; by one on 1/4" and 8 mm cartridge tapes.

1/2" Reel Tape

Data bytes are recorded in parallel onto the 9–track tape. The number of bytes in a physical record varies between 1 and 65535 bytes.

The recording formats available (check specific tape drive) are 800 BPI, 1600 BPI, and 6250 BPI, and data compression. Actual storage capacity is a function of the recording format and the length of the tape reel. For example, using a 2400 foot tape, 20 MB can be stored using 800 BPI, 40 MB using 1600 BPI, 140 MB using 6250 BPI, or up to 700 MB using data compression.

1/4" Cartridge Tape

Data is recorded serially onto 1/4" cartridge tape. The number of bytes per record is determined by the physical record size of the device. The I/O request size must be a multiple of the physical record size of the device. For QIC–11, QIC–24, and QIC–150 tape drives the block size is 512 bytes.

The records are recorded on tracks in a serpentine motion. As one track is completed, the drive switches to the next and begins writing in the opposite direction, eliminating the wasted motion of rewinding. Each file, including the last, ends with one file mark.

Storage capacity is based on the number of tracks the drive is capable of recording. For example, 4–track drives can only record 20 MB of data on a 450 foot tape; 9–track drives can record up to 45 MB of data on a tape of the same length. QIC–11 is the only tape format available for 4–track tape drives. In contrast, 9–track tape drives can use either QIC–24 or QIC–11. Storage capacity is not appreciably affected by using either format. QIC–24 is preferable to QIC–11 because it records a reference signal to mark the position of the first track on the tape, and each block has a unique block number.

The QIC–150 tape drives require DC–6150 (or equivalent) tape cartridges for writing. However, they can read other tape cartridges in QIC–11, QIC–24, QIC–120, or QIC–150 tape formats.

8 mm Cartridge Tape

Data is recorded serially onto 8 mm helical scan cartridge tape. The number of bytes in a physical record varies between 1 and 65535 bytes. Currently one density is available.

Read Operation

read(2V) reads the next record on the tape. The record size is passed back as the number of bytes read, provided it is no greater than the number requested. When a tape mark is read, a zero byte count is returned; another read will fetch the first record of the next tape file. Two successive reads returning zero byte counts indicate the EOM. No further reading should be performed past the EOM.

Fixed-length I/O tape devices require the number of bytes read to be a multiple of the physical record size. For example, 1/4" cartridge tape devices only read multiples of 512 bytes. If the blocking factor is greater than 64512 bytes (minphys limit), fixed-length I/O tape devices read multiple records.

Tape devices which support variable-length I/O operations, such as 1/2" and 8mm tape, may read a range of 1 to 65535 bytes. If the record size exceeds 65535 bytes, the driver reads multiple records to satisfy the request. These multiple records are limited to 65534 bytes.

Write Operation

write(2V) writes the next record on the tape. The record has the same length as the given buffer.

Writing is allowed on 1/4" tape at either the beginning of tape or after the last written file on the tape.

Writing is not so restricted on 1/2" and 8 mm cartridge tape. Care should be used when appending files onto 1/2" reel tape devices, since an extra file mark is appended after the last file to mark the EOM. This extra file mark must be overwritten to prevent the creation of a null file. To facilitate write append operations, a space to the EOM ioctl is provided. Care should be taken when overwriting records; the erase head is just forward of the write head and any following records will also be erased.

Fixed-length I/O tape devices require the number of bytes written to be a multiple of the physical record size. For example, 1/4" cartridge tape devices only write multiples of 512 bytes. Fixed-length I/O tape devices write multiple records if the blocking factor is greater than 64512 bytes (minphys limit). These multiple writes are limited to 64512 bytes. For example, if a write request is issued for 65536 bytes using a 1/4" cartridge tape, two writes are issued; the first for 64512 bytes and the second for 1024 bytes.

Tape devices which support variable-length I/O operations, such as 1/2" and 8mm tape, may write a range of 1 to 65535 bytes. If the record size exceeds 65535 bytes, the driver writes multiple records to satisfy the request. These multiple records are limited to 65534 bytes. As an example, if a write request for 65540 bytes is issued using 1/2" reel tape, two records are written; one for 65534 bytes followed by one for 6 bytes.

EOT handling on write is different among the various devices; see the appropriate device manual page. Reading past EOT is transparent to the user.

Seeks are ignored in tape I/O.

Close Operation

Magnetic tapes are rewound when closed, except when the "no-rewind" devices have been specified. The names of no-rewind device files use the letter **n** as the beginning of the final component. The no-rewind version of **/dev/rmt0** is **/dev/nrmt0**.

If data was written, a file mark is automatically written by the driver upon close. If the rewinding device was specified, the tape will be rewound after the file mark is written. If the user wrote a file mark prior to closing, then no file mark is written upon close. If a file positioning ioctl, like rewind, is issued after writing, a file mark is written before repositioning the tape.

Note: for 1/2" reel tape devices, two file marks are written to mark the EOM before rewinding or performing a file positioning ioctl. If the user wrote a file mark before closing a 1/2" reel tape device, the driver will always write a file mark before closing to insure that the end of recorded media is marked properly. If the non-rewinding **xt** device was specified, two file marks are written and the tape is left positioned between the two so that the second one is overwritten on a subsequent **open**(2V) and **write**(2V). For performance reasons, the **st** driver postpones writing the second tape mark until just before a file positioning ioctl is issued (for example, rewind). This means that the user must not manually rewind the tape because the tape will be missing the second tape mark which marks EOM.

If no data was written and the driver was opened for WRITE-ONLY access, a file mark is written thus creating a null file.

Ioctls

Not all devices support all ioctls. The driver returns an ENOTTY error on unsupported ioctls.

The following structure definitions for magnetic tape ioctl commands are from **<sys/mtio.h>**:

```
/* structure for MTIOCTOP - magnetic tape operation command */
struct  mtop  {
        short   mt_op;          /* operation */
        daddr_t mt_count;       /* number of operations */
};
```

The following ioctls are supported:

MTWEOF	write an end-of-file record
MTFSF	forward space over file mark
MTBSF	backward space over file mark (1/2", 8 mm only)
MTFSR	forward space to inter-record gap
MTBSR	backward space to inter-record gap
MTREW	rewind
MTOFFL	rewind and take the drive offline
MTNOP	no operation, sets status only
MTRETEN	retension the tape (cartridge tape only)
MTERASE	erase the entire tape and rewind
MTEOM	position to EOM
MTNBSF	backward space file to beginning of file

```
/* structure for MTIOCGET - magnetic tape get status command */
struct  mtget {
        short   mt_type;                /* type of magtape device */
/* the following two registers are device dependent */
        short   mt_dsreg;               /* "drive status" register */
        short   mt_erreg;               /* "error" register */
/* optional error info. */
        daddr_t mt_resid;               /* residual count */
        daddr_t mt_fileno;              /* file number of current position */
        daddr_t mt_blkno;               /* block number of current position */
        u_short mt_flags;
        short   mt_bf;                  /* optimum blocking factor */
};
```

When spacing forward over a record (either data or EOF), the tape head is positioned in the tape gap between the record just skipped and the next record. When spacing forward over file marks (EOF records), the tape head is positioned in the tape gap between the next EOF record and the record that follows it.

When spacing backward over a record (either data or EOF), the tape head is positioned in the tape gap immediately preceding the tape record where the tape head is currently positioned. When spacing backward over file marks (EOF records), the tape head is positioned in the tape gap preceding the EOF. Thus the next read would fetch the EOF.

Note, the following features are unique to the st driver: record skipping does not go past a file mark; file skipping does not go past the EOM. Both the st and xt drivers stop upon encountering EOF during a record skipping command, but leave the tape positioned differently. For example, after an MTFSR <huge number> command the st driver leaves the tape positioned *before* the EOF. After the same command, the xt driver leaves the tapes positioned *after* the EOF. Consequently on the next read, the xt driver fetches the first record of the next file whereas the st driver fetches the EOF. A related st feature is that EOFs remain pending until the tape is closed. For example, a program which first reads all the records of a file up to and including the EOF and then performs an MTFSF command will leave the tape positioned just after that same EOF, rather than skipping the next file.

The MTNBSF and MTFSF operations are inverses. Thus, an MTFSF "–1" is equivalent to an MTNBSF "1". An MTNBSF "0" is the same as MTFSF "0"; both position the tape device to the beginning of the current file.

MTBSF moves the tape backwards by file marks. The tape position will end on the beginning of tape side of the desired file mark.

MTBSR and MTFSR operations perform much like space file operations, except that they move by records instead of files. Variable-length I/O devices (1/2" reel, for example) space actual records; fixed-length I/O devices space physical records (blocks). 1/4" cartridge tape, for example, spaces 512 byte physical records. The status ioctl residual count contains the number of files or records not skipped.

MTOFFL rewinds and, if appropriate, takes the device offline by unloading the tape. The tape must be inserted before the tape device can be used again.

MTRETEN The retension ioctl only applies to 1/4" cartridge tape devices. It is used to restore tape tension improving the tape's soft error rate after extensive start-stop operations or long-term storage.

MTERASE rewinds the tape, erases it completely, and returns to the beginning of tape.

MTEOM positions the tape at a location just after the last file written on the tape. For 1/4" cartridge and 8 mm tape, this is after the last file mark on the tape. For 1/2" reel tape, this is just after the first file mark but before the second (and last) file mark on the tape. Additional files can then be appended onto the tape from that point.

Note the difference between MTBSF (backspace over file mark) and MTNBSF (backspace file to beginning of file). The former moves the tape backward until it crosses an EOF mark, leaving the tape positioned *before* the file mark. The latter leaves the tape positioned *after* the file mark. Hence, "MTNBSF n" is equivalent to "MTBSF (n+1)" followed by "MTFSF 1". 1/4 " cartridge tape devices do not support MTBSF.

The MTIOCGET get status ioctl call returns the drive id (*mt_type*), sense key error (*mt_erreg*), file number (*mt_fileno*), optimum blocking factor (*mt_bf*) and record number (*mt_blkno*) of the last error. The residual count (*mt_resid*) is set to the number of bytes not transferred or files/records not spaced. The flags word (*mt_flags*) contains information such as whether the device is SCSI, whether it is a reel device and whether the device supports absolute file positioning.

EXAMPLES

Suppose you have written 3 files to the non-rewinding 1/2" tape device, **/dev/nrmt0**, and that you want to go back and **dd**(1) the second file off the tape. The commands to do this are:

```
mt –f /dev/nrmt0 bsf 3
mt –f /dev/nrmt0 fsf 1
dd if=/dev/nrmt0
```

To accomplish the same tape positioning in a C program, followed by a get status ioctl:

```
struct mtop mt_command;
struct mtget mt_status;

mt_command.mt_op = MTBSF;
mt_command.mt_count = 3;
ioctl(fd, MTIOCTOP, &mt_command);
mt_command.mt_op = MTFSF;
mt_command.mt_count = 1;
ioctl(fd, MTIOCTOP, &mt_command);
ioctl(fd, MTIOCGET, (char *)&mt_status);
```

or

```
struct mtop mt_command;
struct mtget mt_status;

mt_command.mt_op = MTNBSF;
mt_command.mt_count = 2;
ioctl(fd, MTIOCTOP, &mt_command);
ioctl(fd, MTIOCGET, (char *)&mt_status);
```

FILES

/dev/rmt*
/dev/rst*
/dev/rar*
/dev/nrmt*
/dev/nrst*
/dev/nrar*

SEE ALSO

dd(1), **mt**(1), **tar**(1), **read**(2V), **write**(2V), **ar**(4S), **st**(4S), **tm**(4S), **xt**(4S)

1/4 Inch Tape Drive Tutorial

WARNINGS

Avoid the use of device files **/dev/rmt4** and **/dev/rmt12**, as they are going away in a future release.

NAME

nfs, NFS – network file system

CONFIG

options NFS

DESCRIPTION

The Network File System, or NFS, allows a client workstation to perform transparent file access over the network. Using it, a client workstation can operate on files that reside on a variety of servers, server architectures and across a variety of operating systems. Client file access calls are converted to NFS protocol requests, and are 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.

The Network File System operates in a stateless fashion using remote procedure (RPC) calls built on top of external data representation (XDR) protocol. These protocols are documented in *Network Programming* The RPC protocol provides for version and authentication parameters to be exchanged for security over the network.

A server can grant access to a specific filesystem to certain clients by adding an entry for that filesystem to the server's **/etc/exports** file and running **exportfs**(8).

A client gains access to that filesystem with the **mount**(2V) system call, which requests a file handle for the filesystem itself. Once the filesystem is mounted by the client, the server issues a file handle to the client for each file (or directory) the client accesses or creates. If the file is somehow removed on the server side, the file handle becomes stale (dissociated with a known file).

A server may also be a client with respect to filesystems it has mounted over the network, but its clients cannot gain access to those filesystems. Instead, the client must mount a filesystem directly from the server on which it resides.

The user ID and group ID mappings must be the same between client and server. However, the server maps uid 0 (the super-user) to uid −2 before performing access checks for a client. This inhibits super-user privileges on remote filesystems. This may be changed by use of the "anon" export option. See **exportfs**(8).

NFS-related routines and structure definitions are described in *Network Programming*.

ERRORS

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 will see the console message:

NFS server *host* **not responding still trying.**

Depending on whether the file system has been mounted "hard" or "soft" (see **mount**(8)), the client will either continue (forever) to resend the request until it receives an acknowledgement from the server, or return an error to user-level. For hard mounts, this means the server can crash or power down and come back up without any special action required by the client. If the "intr" mount option was not specified, a client process requesting I/O will block and remain insensitive to signals, sleeping inside the kernel at **PRIBIO** until the request is satisfied.

FILES

/etc/exports

SEE ALSO

mount(2V), **exports**(5), **fstab**(5), **fstab**(5), **exportfs**(8), **mount**(8), **nfsd**(8), **sticky**(8)

Network Programming

BUGS

When a file that is opened by a client is unlinked (by the server), a file with a name of the form **.nfs***XXX* (where *XXX* is a number) is created by the client. When the open file is closed, the **.nfs***XXX* file is removed. If the client crashes before the file can be closed, the **.nfs***XXX* file is not removed.

NFS servers usually mark their clients' swap files specially to avoid being required to sync their inodes to disk before returning from writes. See **sticky**(8).

NAME

nit – Network Interface Tap

CONFIG

```
pseudo-device   clone
pseudo-device   snit
pseudo-device   pf
pseudo-device   nbuf
```

SYNOPSIS

```
#include <sys/file.h>
#include <sys/ioctl.h>
#include <net/nit_pf.h>
#include <net/nit_buf.h>

        fd = open("/dev/nit", mode);
        ioctl(fd, I_PUSH, "pf");
        ioctl(fd, I_PUSH, "nbuf");
```

DESCRIPTION

NIT (the Network Interface Tap) is a facility composed of several STREAMS modules and drivers. These components collectively provide facilities for constructing applications that require link-level network access. Examples of such applications include **rarpd**(8C), which is a user-level implementation of the Reverse ARP protocol, and **etherfind**(8C), which is a network monitoring and trouble-shooting program.

NIT consists of several components that are summarized below. See their Reference Manual entries for detailed information about their specification and operation.

nit_if(4M) This component is a STREAMS device driver that interacts directly with the system's Ethernet drivers. After opening an instance of this device it must be bound to a specific Ethernet interface before becoming usable. Subsequently, **nit_if** transcribes packets arriving on the interface to the read side of its associated stream and delivers messages reaching it on the write side of its stream to the raw packet output code for transmission over the interface.

nit_pf(4M) This module provides packet-filtering services, allowing uninteresting incoming packets to be discarded with minimal loss of efficiency. It passes through unaltered all outgoing messages (those on the stream's write side).

nit_buf(4M) This module buffers incoming messages into larger aggregates, thereby reducing the overhead incurred by repeated **read**(2V) system calls.

NIT clients mix and match these components, based on their particular requirements. For example, the reverse ARP daemon concerns itself only with packets of a specific type and deals with low traffic volumes. Thus, it uses **nit_if** for access to the network and **nit_pf** to filter out all incoming packets except reverse ARP packets, but omits the **nit_buf** buffering module since traffic is not high enough to justify the additional complexity of unpacking buffered packets. On the other hand, the **etherd**(8C) program, which collects Ethernet statistics for **traffic**(1C) to display, must examine every packet on the network. Therefore, it omits the **nit_pf** module, since there is nothing it wishes to screen out, and includes the **nit_buf** module, since most networks have very heavy aggregate packet traffic.

EXAMPLES

The following code fragments outline how to program against parts of the NIT interface. For the sake of brevity, all error-handling code has been elided.

initdevice comes from **etherfind** and sets up its input stream configuration.

```
initdevice(if_flags, snaplen, chunksize)
        u_long  if_flags,
                snaplen,
                chunksize;
```

```
{
        struct strioctl     si;
        struct ifreq        ifr;
        struct timeval      timeout;

        if_fd = open(NIT_DEV, O_RDONLY);

        /* Arrange to get discrete messages from the stream. */
        ioctl(if_fd, I_SRDOPT, (char *)RMSGD);

        si.ic_timout = INFTIM;

        /* Push and configure the buffering module. */
        ioctl(if_fd, I_PUSH, "nbuf");

        timeout.tv_sec = 1;
        timeout.tv_usec = 0;
        si.ic_cmd = NIOCSTIME;
        si.ic_len = sizeof timeout;
        si.ic_dp = (char *)&timeout;
        ioctl(if_fd, I_STR, (char *)&si);

        si.ic_cmd = NIOCSCHUNK;
        si.ic_len = sizeof chunksize;
        si.ic_dp = (char *)&chunksize;
        ioctl(if_fd, I_STR, (char *)&si);

        /* Configure the nit device, binding it to the proper
          underlying interface, setting the snapshot length,
          and setting nit_if-level flags. */
        strncpy(ifr.ifr_name, device, sizeof ifr.ifr_name);
        ifr.ifr_name[sizeof ifr.ifr_name - 1] = '\0';
        si.ic_cmd = NIOCBIND;
        si.ic_len = sizeof ifr;
        si.ic_dp = (char *)&ifr;
        ioctl(if_fd, I_STR, (char *)&si);

        if (snaplen > 0) {
                si.ic_cmd = NIOCSSNAP;
                si.ic_len = sizeof snaplen;
                si.ic_dp = (char *)&snaplen;
                ioctl(if_fd, I_STR, (char *)&si);
        }

        if (if_flags != 0) {
                si.ic_cmd = NIOCSFLAGS;
                si.ic_len = sizeof if_flags;
                si.ic_dp = (char *)&if_flags;
                ioctl(if_fd, I_STR, (char *)&si);
        }

        /* Flush the read queue, to get rid of anything that accumulated
          before the device reached its final configuration. */
        ioctl(if_fd, I_FLUSH, (char *)FLUSHR);
}
```

Here is the skeleton of the packet reading loop from **etherfind**. It illustrates how to cope with dismantling the headers the various NIT components glue on.

```
while ((cc = read(if_fd, buf, chunksize)) >= 0) {
        register u_char     *bp = buf,
                            *bufstop = buf + cc;

        /* Loop through each message in the chunk. */
        while (bp < bufstop) {
                register u_cbar             *cp = bp;
                struct nit_bufhdr  *bdrp;
                struct timeval              *tvp = NULL;
                u_long                      drops = 0;
                u_long                      pktlen;

                /* Extract information from the successive objects
                  embedded in the current message. Which ones we
                  have depends on how we set up the stream (and
                  therefore on what command line flags were set).

                  If snaplen is positive then the packet was truncated
                  before the buffering module saw it, so we must
                  obtain its length from the nit_if-level nit_iflen
                  header. Otherwise the value in *hdrp suffices. */
                hdrp = (struct nit_bufhdr *)cp;
                cp += sizeof *hdrp;
                if (tflag) {
                        struct nit_iftime   *ntp;

                        ntp = (struct nit_iftime *)cp;
                        cp += sizeof *ntp;

                        tvp = &ntp->nh_timestamp;
                }
                if (dflag) {
                        struct nit_ifdrops  *ndp;

                        ndp = (struct nit_ifdrops *)cp;
                        cp += sizeof *ndp;

                        drops = ndp->nh_drops;
                }
                if (snaplen > 0) {
                        struct nit_iflen     *nlp;

                        nlp = (struct nit_iflen *)cp;
                        cp += sizeof *nlp;

                        pktlen = nlp->nh_pktlen;
                }
                else
                        pktlen = bdrp->nhb_msglen;

                sp = (struct sample *)cp;
                bp += hdrp->nhb_totlen;

                /* Process the packet. */
        }
}
```

FILES

/dev/nit	clone device instance referring to **nit_if**

SEE ALSO

traffic(1C), **read**(2V), **nit_if**(4M), **nit_pf**(4M), **nit_buf**(4M), **etherd**(8C), **etherfind**(8C), **rarpd**(8C)

NAME

nit_buf – STREAMS NIT buffering module

CONFIG

pseudo-device nbuf

SYNOPSIS

```
#include <sys/ioctl.h>
#include <net/nit_buf.h>
ioctl(fd, I_PUSH, "nbuf");
```

DESCRIPTION

nit_buf is a STREAMS module that buffers incoming messages, thereby reducing the number of system calls and associated overhead required to read and process them. Although designed to be used in conjunction with the other components of NIT (see **nit**(4P)), **nit_buf** is a general-purpose module and can be used anywhere STREAMS input buffering is required.

Read-side Behavior

nit_buf collects incoming M_DATA and M_PROTO messages into *chunks*, passing each chunk upward when either the chunk becomes full or the current read timeout expires. When a message arrives, it is processed in two steps. First, the message is prepared for inclusion in a chunk, and then it is added to the current chunk. The following paragraphs discuss each step in turn.

Upon receiving a message from below, **nit_buf** immediately converts all leading M_PROTO blocks in the message to M_DATA blocks, altering only the message type field and leaving the contents alone. It then prepends a header to the converted message. This header is defined as follows.

```
struct nit_bufhdr {
        u_int   nhb_msglen;
        u_int   nhb_totlen;
};
```

The first field of this header gives the length in bytes of the converted message. The second field gives the distance in bytes from the start of the message in the current chunk (described below) to the start of the next message in the chunk; the value reflects any padding necessary to insure correct data alignment for the host machine and includes the length of the header itself.

After preparing a message, **nit_buf** attempts to add it to the end of the current chunk, using the chunk size and timeout values to govern the addition. (The chunk size and timeout values are set and inspected using the **ioctl** calls described below.) If adding the new message would make the current chunk grow larger than the chunk size, **nit_buf** closes off the current chunk, passing it up to the next module in line, and starts a new chunk, seeding it with a zero-length message. If adding the message would still make the current chunk overflow, the module passes it upward in an over-size chunk of its own. Otherwise, the module concatenates the message to the end of the current chunk.

To ensure that messages do not languish forever in an accumulating chunk, **nit_buf** maintains a read timeout. Whenever this timeout expires, the module closes off the current chunk, regardless of its length, and passes it upward; if no incoming messages have arrived, the chunk passed upward will have zero length. Whenever the module passes a chunk upward, it restarts the timeout period. These two rules insure that **nit_buf** minimizes the number of chunks it produces during periods of intense message activity and that it periodically disposes of all messages during slack intervals.

nit_buf handles other message types as follows. Upon receiving an M_FLUSH message specifying that the read queue be flushed, the module does so, clearing the currently accumulating chunk as well, and passes the message on to the module or driver above. It passes all other messages through unaltered to its upper neighbor.

Write-side Behavior

nit_buf intercepts M_IOCTL messages for the *ioctl*s described below. Upon receiving an M_FLUSH message specifying that the write queue be flushed, the module does so and passes the message on to the module or driver below. The module passes all other messages through unaltered to its lower neighbor.

IOCTLS

nit_buf responds to the following *ioctl*s.

NIOCSTIME Set the read timeout value to the value referred to by the *struct timeval* pointer given as argument. Setting the timeout value to zero has the side-effect of forcing the chunk size to zero as well, so that the module will pass all incoming messages upward immediately upon arrival.

NIOCGTIME Return the read timeout in the *struct timeval* pointed to by the argument. If the timeout has been cleared with the **NIOCCTIME** *ioctl*, return with an ERANGE error.

NIOCCTIME Clear the read timeout, effectively setting its value to infinity.

NIOCSCHUNK Set the chunk size to the value referred to by the *u_int* pointer given as argument.

NIOCGCHUNK Return the chunk size in the *u_int* pointed to by the argument.

WARNING

The module name "nbuf" used in the system configuration file and as argument to the **I_PUSH ioctl** is provisional and subject to change.

SEE ALSO

nit(4P), **nit_if**(4M), **nit_pf**(4M)

NAME

nit_if – STREAMS NIT device interface module

CONFIG

pseudo-device snit

SYNOPSIS

```
#include <sys/file.h>
open("/dev/nit", mode);
```

DESCRIPTION

nit_if is a STREAMS pseudo-device driver that provides STREAMS access to network interfaces. It is designed to be used in conjunction with the other components of NIT (see **nit**(4P)), but can be used by itself as a raw STREAMS network interface.

nit_if is an exclusive-open device that is intended to be opened indirectly through the clone device; **/dev/nit** is a suitable instance of the clone device. Before the stream resulting from opening an instance of **nit_if** may be used to read or write packets, it must first be bound to a specific network interface, using the **NIOCSBIND** ioctl described below.

Read-side Behavior

nit_if copies leading prefixes of selected packets from its associated network interface and passes them up the stream. If the **NI_PROMISC** flag is set, it passes along all packets; otherwise it passes along only packets addressed to the underlying interface.

The amount of data copied from a given packet depends on the current *snapshot length*, which is set with the **NIOCSSNAP** ioctl described below.

Before passing each packet prefix upward, **nit_if** optionally prepends one or more headers, as controlled by the state of the flag bits set with the **NIOCSFLAGS** *ioctl*. The driver collects headers into **M_PROTO** message blocks, with the headers guaranteed to be completely contained in a single message block, whereas the packet itself goes into one or more **M_DATA** message blocks.

Write-side Behavior

nit_if accepts packets from the module above it in the stream and relays them to the associated network interface for transmission. Packets must be formatted with the destination address in a leading **M_PROTO** message block, followed by the packet itself, complete with link-level header, in a sequence of **M_DATA** message blocks. The destination address must be expressed as a '**struct sockaddr**' whose *sa_family* field is **AF_UNSPEC** and whose *sa_data* field is a copy of the link-level header. (See **sys/socket.h** for the definition of this structure.) If the packet does not conform to this format, an **M_ERROR** message with EINVAL will be sent upstream.

nit_if processes **M_IOCTL** messages as described below. Upon receiving an **M_FLUSH** message specifying that the write queue be flushed, **nit_if** does so and transfers the message to the read side of the stream. It discards all other messages.

IOCTLS

nit_if responds to the following *ioctl*s, as defined in **net/nit_if.h**. It generates an **M_IOCNAK** message for all others, returning this message to the invoker along the read side of the stream.

SIOCGIFADDR

SIOCADDMULTI

SIOCDELMULTI **nit_if** passes these ioctls on to the underlying interface's driver and returns its response in a '**struct ifreq**' instance, as defined in **net/if.h**. (See the description of this ioctl in **if**(4N) for more details.)

NIOCBIND This ioctl attaches the stream represented by its first argument to the network interface designated by its third argument, which should be a pointer to an *ifreq* structure whose *ifr_name* field names the desired interface. See **net/if.h** for the definition of this structure.

NIOCSSNAP Set the current snapshot length to the value given in the *u_long* pointed to by the *ioctl*'s final argument. **nit_if** interprets a snapshot length value of zero as meaning infinity, so that it will copy all selected packets in their entirety. It constrains positive snapshot lengths to be at least the length of an Ethernet header, so that it will pass at least the link-level header of all selected packets to its upstream neighbor.

NIOCGSNAP Returns the current snapshot length for this device instance in the *u_long* pointed to by the *ioctl*'s final argument.

NIOCSFLAGS **nit_if** recognizes the following flag bits, which must be given in the *u_long* pointed to by the *ioctl*'s final argument. This set may be augmented in future releases. All but the **NI_PROMISC** bit control the addition of headers that precede the packet body. These headers appear in the order given below, with the last-mentioned enabled header adjacent to the packet body.

NI_PROMISC Requests that the underlying interface be set into promiscuous mode and that all packets that the interface receives be passed up through the stream. **nit_if** only honors this bit for the super-user.

NI_TIMESTAMP Prepend to each selected packet a header containing the packet arrival time expressed as a '**struct timeval**'.

NI_DROPS Prepend to each selected packet a header containing the cumulative number of packets that this instance of **nit_if** has dropped because of flow control requirements or resource exhaustion. The header value is expressed as a *u_long*. Note: it accounts only for events occurring within **nit_if**, and does not count packets dropped at the network interface level or by upstream modules.

NI_LEN Prepend to each selected packet a header containing the packet's original length (including link-level header), as it was before being trimmed to the snapshot length. The header value is expressed as a *u_long*.

NIOCGFLAGS Returns the current state of the flag bits for this device instance in the *u_long* pointed to by the *ioctl*'s final argument.

FILES

/dev/nit clone device instance referring to **nit_if** device

net/nit_if.h header file containing definitions for the *ioctl*s and packet headers described above.

SEE ALSO

clone(4), **nit**(4P), **nit_buf**(4M), **nit_pf**(4M)

NAME

nit_pf – STREAMS NIT packet filtering module

CONFIG

pseudo-device pf

SYNOPSIS

```
#include <sys/ioctl.h>
#include <net/nit_pf.h>
        ioctl(fd, I_PUSH, "pf");
```

DESCRIPTION

nit_pf is a STREAMS module that subjects messages arriving on its read queue to a packet filter and passes only those messages that the filter accepts on to its upstream neighbor. Such filtering can be very useful for user-level protocol implementations and for networking monitoring programs that wish to view only specific types of events.

Read-side Behavior

nit_pf applies the current packet filter to all M_DATA and M_PROTO messages arriving on its read queue. The module prepares these messages for examination by first skipping over all leading M_PROTO message blocks to arrive at the beginning of the message's data portion. If there is no data portion, **nit_pf** accepts the message and passes it along to its upstream neighbor. Otherwise, the module ensures that the part of the message's data that the packet filter might examine lies in contiguous memory, calling the *pullupmsg* utility routine if necessary to force contiguity. (Note: this action destroys any sharing relationships that the subject message might have had with other messages.) Finally, it applies the packet filter to the message's data, passing the entire message upstream to the next module if the filter accepts, and discarding the message otherwise. See PACKET FILTERS below for details on how the filter works.

If there is no packet filter yet in effect, the module acts as if the filter exists but does nothing, implying that all incoming messages are accepted. IOCTLS below describes how to associate a packet filter with an instance of **nit_pf**.

nit_pf handles other message types as follows. Upon receiving an M_FLUSH message specifying that the read queue be flushed, the module does so, and passes the message on to its upstream neighbor. It passes all other messages through unaltered to its upper neighbor.

Write-side Behavior

nit_pf intercepts M_IOCTL messages for the *ioctl* described below. Upon receiving an M_FLUSH message specifying that the write queue be flushed, the module does so and passes the message on to the module or driver below. The module passes all other messages through unaltered to its lower neighbor.

IOCTLS

nit_pf responds to the following *ioctl*.

NIOCSETF This *ioctl* directs the module to replace its current packet filter, if any, with the filter specified by the '**struct packetfilt**' pointer named by its final argument. This structure is defined in **<net/packetfilt.h>** as

```
struct packetfilt {
    u_char   Pf_Priority;    /* priority of filter */
    u_char   Pf_FilterLen;   /* # of cmds in list */
    u_short  Pf_Filter[ENMAXFILTERS];
                             /* filter command list */
};
```

The *Pf_Priority* field is included only for compatibility with other packet filter implementations and is otherwise ignored. The packet filter itself is specified in the *Pf_Filter* array as a sequence of two-byte commands, with the *Pf_FilterLen* field giving the number of commands in the sequence. This implementation restricts the maximum number of commands in a filter (**ENMAXFILTERS**) to 40. The next section describes the available commands and their semantics.

PACKET FILTERS

A packet filter consists of the filter command list length (in units of *u_shorts*), and the filter command list itself. (The priority field mentioned above is ignored in this implementation.) Each filter command list specifies a sequence of actions that operate on an internal stack of *u_shorts* ("shortwords"). Each shortword of the command list specifies one of the actions ENF_PUSHLIT, ENF_PUSHZERO, or ENF_PUSHWORD+*n*, which respectively push the next shortword of the command list, zero, or shortword *n* of the subject message on the stack, and a binary operator from the set { ENF_EQ, ENF_NEQ, ENF_LT, ENF_LE, ENF_GT, ENF_GE, ENF_AND, ENF_OR, ENF_XOR } which then operates on the top two elements of the stack and replaces them with its result. When both an action and operator are specified in the same shortword, the action is performed followed by the operation.

The binary operator can also be from the set { ENF_COR, ENF_CAND, ENF_CNOR, ENF_CNAND }. These are "short-circuit" operators, in that they terminate the execution of the filter immediately if the condition they are checking for is found, and continue otherwise. All pop two elements from the stack and compare them for equality; ENF_CAND returns false if the result is false; ENF_COR returns true if the result is true; ENF_CNAND returns true if the result is false; ENF_CNOR returns false if the result is true. Unlike the other binary operators, these four do not leave a result on the stack, even if they continue.

The short-circuit operators should be used when possible, to reduce the amount of time spent evaluating filters. When they are used, you should also arrange the order of the tests so that the filter will succeed or fail as soon as possible; for example, checking the IP destination field of a UDP packet is more likely to indicate failure than the packet type field.

The special action ENF_NOPUSH and the special operator ENF_NOP can be used to only perform the binary operation or to only push a value on the stack. Since both are (conveniently) defined to be zero, indicating only an action actually specifies the action followed by ENF_NOP, and indicating only an operation actually specifies ENF_NOPUSH followed by the operation.

After executing the filter command list, a non-zero value (true) left on top of the stack (or an empty stack) causes the incoming packet to be accepted and a zero value (false) causes the packet to be rejected. (If the filter exits as the result of a short-circuit operator, the top-of-stack value is ignored.) Specifying an undefined operation or action in the command list or performing an illegal operation or action (such as pushing a shortword offset past the end of the packet or executing a binary operator with fewer than two shortwords on the stack) causes a filter to reject the packet.

EXAMPLES

The reverse ARP daemon program (**rarpd**(8C)) uses code similar to the following fragment to construct a filter that rejects all but RARP packets. That is, is accepts only packets whose Ethernet type field has the value ETHERTYPE_REVARP.

```
struct ether_header eh;          /* used only for offset values */
struct packetfilt pf;
register u_short *fwp = pf.Pf_Filter;
u_short offset;

/*
 * Set up filter. Offset is the displacement of the Ethernet
 * type field from the beginning of the packet in units of
 * u_shorts.
 */
```

```
offset = ((u_int) &eh.ether_type - (u_int) &eh.ether_dhost) / sizeof (u_short);
*fwp++ = ENF_PUSHWORD + offset;
*fwp++ = ENF_PUSHLIT;
*fwp++ = htons(ETHERTYPE_REVARP);
*fwp++ = ENF_EQ;
pf.Pf_FilterLen = fwp - &pf.Pf_Filter[0];
```

This filter can be abbreviated by taking advantage of the ability to combine actions and operations:

```
...
*fwp++ = ENF_PUSHWORD + offset;
*fwp++ = ENF_PUSHLIT | ENF_EQ;
*fwp++ = htons(ETHERTYPE_REVARP);
...
```

WARNINGS

The module name '**pf**' used in the system configuration file and as argument to the **I_PUSH** *ioctl* is provisional and subject to change.

The *Pf_Priority* field of the *packetfilt* structure is likely to be removed.

SEE ALSO

inet(4F), **nit**(4P), **nit_buf**(4M), **nit_if**(4M)

NAME

null – data sink

CONFIG

None; included with standard system.

SYNOPSIS

#include <fcntl.h>

open("/dev/null", *mode*);

DESCRIPTION

Data written on the **null** special file is discarded.

Reads from the **null** special file always return an end-of-file indication.

FILES

/dev/null

NAME

openprom – PROM monitor configuration interface

CONFIG

pseudo-device openeepr

SYNOPSIS

```
#include <fcntl.h>
#include <sys/types.h>
#include <sundev/openpromio.h>
open("/dev/openprom", mode);
```

AVAILABILITY

SPARCstation 1 systems only.

DESCRIPTION

As with other Sun systems, configuration options are stored in an EEPROM or NVRAM on a SPARCstation 1 system. However, unlike other Sun systems, the encoding of these options is private to the PROM monitor. The **openprom** device provides an interface to the PROM monitor allowing a user program to query and set these configuration options through the use of **ioctl**(2) requests. These requests are defined in **<sundev/openpromio.h>**:

```
struct openpromio {
        u_int   oprom_size;             /* real size of following array */
        char    oprom_array[1];         /* For property names and values */
                                        /* NB: Adjacent, Null terminated */
};
#define OPROMMAXPARAM   1024            /* max size of array */

#define OPROMGETOPT     _IO(O,1)
#define OPROMSETOPT     _IO(O,2)
#define OPROMNXTOPT     _IO(O,3)
```

For all **ioctl**( ) requests, the third parameter is a pointer to a '**struct openpromio**'. All property names and values are null-terminated strings; the value of a numeric option is its ASCII representation.

IOCTLS

The **OPROMGETOPT** ioctl takes the null-terminated name of a property in the *oprom_array* and returns its null-terminated value (overlaying its name). *oprom_size* should be set to the size of *oprom_array*; on return it will contain the size of the returned value. If the named property does not exist, or if there is not enough space to hold its value, then *oprom_size* will be set to zero. See BUGS below.

The **OPROMSETOPT** ioctl takes two adjacent strings in *oprom_array*; the null-terminated property name followed by the null-terminated value.

The **OPROMNXTOPT** ioctl is used to retrieve properties sequentially. The null-terminated name of a property is placed into *oprom_array* and on return it is replaced with the null-terminated name of the next property in the sequence, with *oprom_size* set to its length. A null string on input means return the name of the first property; an *oprom_size* of zero on output means there are no more properties.

ERRORS

EINVAL	The size value was invalid, or (for **OPROMSETOPT**) the property does not exist.
ENOMEM	The kernel could not allocate space to copy the user's structure

FILES

/dev/openprom	PROM monitor configuration interface

SEE ALSO

mem(4S), **eeprom**(8S), **monitor**(8S)

BUGS

There should be separate return values for non-existent properties as opposed to not enough space for the value.

An attempt to set a property to an illegal value results in the PROM setting it to some legal value, with no error being returned. An **OPROMGETOPT** should be performed after an **OPROMSETOPT** to verify that the set worked.

The driver should be more consistent in its treatment of errors and edge conditions.

NAME

pp – Centronics-compatible parallel printer port

CONFIG — Sun386i SYSTEMS

device pp0 at obio ? csr 0x378 irq 15 priority 2

CONFIG — SUN-3x SYSTEMS

device pp0 at obio ? csr 0x6f000000 priority 1

This synopsis line should be used to generate a kernel for Sun-3/80 systems only.

AVAILABILITY

Sun386i and Sun-3/80 systems only.

DESCRIPTION

This device driver provides an interface to the Sun386i and Sun-3/80 systems' on-board Centronics-compatible parallel printer port. It supports most standard PC printers with Centronics interfaces.

FILES

/dev/pp0

DIAGNOSTICS

pp*: printer not online

pp*: printer out of paper

NAME

pty – pseudo-terminal driver

CONFIG

pseudo-device pty*n*

SYNOPSIS

#include <fcntl.h>
#include <sys/termios.h>
open("/dev/ttyp*n*", mode);
open("/dev/ptyp*n*", mode);

DESCRIPTION

The **pty** driver provides support for a pair of devices collectively known as a *pseudo-terminal*. The two devices comprising a pseudo-terminal are known as a *controller* and a *slave*. The slave device distinguishes between the **B0** baud rate and other baud rates specified in the **c_cflag** word of the **termios** structure, and the **CLOCAL** flag in that word. It does not support any of the other **termio**(4) device control functions specified by flags in the **c_cflag** word of the **termios** structure and by the **IGNBRK**, **IGNPAR**, **PARMRK**, or **INPCK** flags in the **c_iflag** word of the **termios** structure, as these functions apply only to asynchronous serial ports. All other **termio**(4) functions must be performed by STREAMS modules pushed atop the driver; when a slave device is opened, the **ldterm**(4M) and **ttcompat**(4M) STREAMS modules are automatically pushed on top of the stream, providing the standard **termio**(4) interface.

Instead of having a hardware interface and associated hardware that supports the terminal functions, the functions are implemented by another process manipulating the controller device of the pseudo-terminal.

The controller and the slave devices of the pseudo-terminal are tightly connected. Any data written on the controller device is given to the slave device as input, as though it had been received from a hardware interface. Any data written on the slave terminal can be read from the controller device (rather than being transmitted from a UART).

In configuring, if no optional "count" is given in the specification, 16 pseudo-terminal pairs are configured.

IOCTLS

The standard set of **termio** **ioctl**s are supported by the slave device. None of the bits in the **c_cflag** word have any effect on the pseudo-terminal, except that if the baud rate is set to **B0**, it will appear to the process on the controller device as if the last process on the slave device had closed the line; thus, setting the baud rate to **B0** has the effect of "hanging up" the pseudo-terminal, just as it has the effect of "hanging up" a real terminal.

There is no notion of "parity" on a pseudo-terminal, so none of the flags in the **c_iflag** word that control the processing of parity errors have any effect. Similarly, there is no notion of a "break", so none of the flags that control the processing of breaks, and none of the **ioctl**s that generate breaks, have any effect.

Input flow control is automatically performed; a process that attempts to write to the controller device will be blocked if too much unconsumed data is buffered on the slave device. The input flow control provided by the **IXOFF** flag in the **c_iflag** word is not supported.

The delays specified in the **c_oflag** word are not supported.

As there are no modems involved in a pseudo-terminal, the **ioctl**s that return or alter the state of modem control lines are silently ignored.

On Sun systems, an additional **ioctl** is provided:

TIOCCONS

The argument is ignored. All output that would normally be sent to the console (either from programs writing to **/dev/console** or from kernel printouts) is redirected so that it is written to the pseudo-terminal instead.

A few special **ioctl**s are provided on the controller devices of pseudo-terminals to provide the functionality needed by applications programs to emulate real hardware interfaces:

TIOCSTOP
The argument is ignored. Output to the pseudo-terminal is suspended, as if a **STOP** character had been typed.

TIOCSTART
The argument is ignored. Output to the pseudo-terminal is restarted, as if a **START** character had been typed.

TIOCPKT
The argument is a pointer to an **int**. If the value of the **int** is non-zero, *packet* mode is enabled; if the value of the **int** is zero, packet mode is disabled. When a pseudo-terminal is in packet mode, each subsequent **read**(2V) from the controller device will return data written on the slave device preceded by a zero byte (symbolically defined as **TIOCPKT_DATA**), or a single byte reflecting control status information. In the latter case, the byte is an inclusive-or of zero or more of the bits:

TIOCPKT_FLUSHREAD
whenever the read queue for the terminal is flushed.

TIOCPKT_FLUSHWRITE
whenever the write queue for the terminal is flushed.

TIOCPKT_STOP whenever output to the terminal is stopped using ^S.

TIOCPKT_START whenever output to the terminal is restarted.

TIOCPKT_DOSTOP whenever XON/XOFF flow control is enabled after being disabled; it is considered "enabled" when the **IXON** flag in the **c_iflag** word is set, the **VSTOP** member of the **c_cc** array is ^S and the **VSTART** member of the **c_cc** array is ^Q.

TIOCPKT_NOSTOP whenever XON/XOFF flow control is disabled after being enabled.

This mode is used by **rlogin**(1C) and **rlogind**(8C) to implement a remote-echoed, locally ^S/^Q flow-controlled remote login with proper back-flushing of output when interrupts occur; it can be used by other similar programs.

TIOCREMOTE
The argument is a pointer to an **int**. If the value of the **int** is non-zero, *remote* mode is enabled; if the value of the **int** is zero, remote mode is disabled. This mode can be enabled or disabled independently of packet mode. When a pseudo-terminal is in remote mode, input to the slave device of the pseudo-terminal is flow controlled and not input edited (regardless of the mode the slave side of the pseudo-terminal). Each write to the controller device produces a record boundary for the process reading the slave device. In normal usage, a write of data is like the data typed as a line on the terminal; a write of 0 bytes is like typing an **EOF** character. Note: this means that a process writing to a pseudo-terminal controller in *remote* mode must keep track of line boundaries, and write only one line at a time to the controller. If, for example, it were to buffer up several **NEWLINE** characters and write them to the controller with one **write**(), it would appear to a process reading from the slave as if a single line containing several **NEWLINE** characters had been typed (as if, for example, a user had typed the **LNEXT** character before typing all but the last of those **NEWLINE** characters). Remote mode can be used when doing remote line editing in a window manager, or whenever flow controlled input is required.

The **ioctl**s **TIOCGWINSZ**, **TIOCSWINSZ**, and, on Sun systems, **TIOCCONS**, can be performed on the controller device of a pseudo-terminal; they have the same effect as when performed on the slave device.

FILES

/dev/pty[p-s][0-9a-f]	pseudo-terminal controller devices
/dev/tty[p-s][0-9a-f]	pseudo-terminal slave devices
/dev/console	

SEE ALSO

rlogin(1C), **termio**(4), **ldterm**(4M), **ttcompat**(4M), **rlogind**(8C)

BUGS

It is apparently not possible to send an **EOT** by writing zero bytes in **TIOCREMOTE** mode.

NAME

rfs, RFS – remote file sharing

CONFIGURATION

options RFS
options VFSSTATS

AVAILABILITY

Available only with the *RFS* software installation option. Refer to *Installing SunOS 4.1* for information on how to install optional software.

DESCRIPTION

The Remote File Sharing service, or RFS, allows transparent resource sharing among hosts on a network. A *resource* can be a directory, the files contained in that directory, subdirectories, devices, and even named pipes. Resources are advertised as a local directory using the name services. Hosts can then mount these resources, and use them as they would a local file system. The host advertising the resource is a file server, the hosts mounting the resource are clients.

All file servers and clients on a network belong to an RFS *domain*, and are administered by the same RFS name server. A domain consists of the following:

- A primary name server
- Possibly one or more secondary name servers
- File servers
- Clients

The name server maintains a list of advertised resources, and passwords in use. The name server also provides *name-to-resource* mapping. This allows a client to mount an advertised resource by the resource name, without needing to know the name of the file server or the pathname of the directory.

FILES

/usr/nserve/rfmaster hosts providing domain name service

SEE ALSO

clone(4), **nit_buf**(4M), **nit_pm**(4M), **tcptli**(4P), **timod**(4), **tirdwr**(4), **rfadmin**(8), **rfstart**(8), **rfudaemon**(8), **rmntstat**(8)

System and Network Administration

NAME

root – pseudo-driver for Sun386i root disk

CONFIG

pseudo-device rootdev

AVAILABILITY

Available only on Sun 386i systems running a SunOS 4.0*x* release or earlier. Not a SunOS 4.1 release feature.

DESCRIPTION

The **root** pseudo-driver provides indirect, device-independent access to the root disk on a diskful Sun workstation. The root disk is the disk where the mounted root partition resides - typically the disk from which the system was booted.

The intent of the **root** device is to allow uniform access to the partitions on the root disk, regardless of the disk's controller type or unit number. For example, the following version of **/etc/fstab** will work for any disk (assuming the disk has the standard partitions and filesystems):

```
/dev/roota /      4.2 rw 1 1
/dev/rootg /usr   4.2 ro 1 2
/dev/rooth /export 4.2 rw 1 3
```

When the root device is opened, the open and all subsequent operations on that device (**read**(2V), **write**(2V), **ioctl**(2), **close**(2V)) are redirected to the real disk. Therefore, all device-dependent operations on a particular disk are still accessible via the root device (see **dkio**(4S)).

FILES

/dev/root[a-h]	block partitions
/dev/rroot[a-h]	raw partitions

SEE ALSO

fstab(5), **sd**(4S), **open**(2V), **dkio**(4S)

NAME

routing – system supporting for local network packet routing

DESCRIPTION

The network facilities provided general packet routing, leaving routing table maintenance to applications processes.

A simple set of data structures comprise a "routing table" used in selecting the appropriate network interface when transmitting packets. This table contains a single entry for each route to a specific network or host. A user process, the routing daemon, maintains this data base with the aid of two socket specific **ioctl**(2) commands, **SIOCADDRT** and **SIOCDELRT**. The commands allow the addition and deletion of a single routing table entry, respectively. Routing table manipulations may only be carried out by super-user.

A routing table entry has the following form, as defined in **<net/route.h>**:

```
struct rtentry {
        u_long  rt_hash;
        struct  sockaddr rt_dst;
        struct  sockaddr rt_gateway;
        short   rt_flags;
        short   rt_refcnt;
        u_long  rt_use;
        struct  ifnet *rt_ifp;
};
```

with *rt_flags* defined from:

```
#define RTF_UP          0x1             /* route usable */
#define RTF_GATEWAY 0x2                 /* destination is a gateway */
#define RTF_HOST        0x4             /* host entry (net otherwise) */
```

Routing table entries come in three flavors: for a specific host, for all hosts on a specific network, for any destination not matched by entries of the first two types (a wildcard route). When the system is booted, each network interface autoconfigured installs a routing table entry when it wishes to have packets sent through it. Normally the interface specifies the route through it is a "direct" connection to the destination host or network. If the route is direct, the transport layer of a protocol family usually requests the packet be sent to the same host specified in the packet. Otherwise, the interface may be requested to address the packet to an entity different from the eventual recipient (that is, the packet is forwarded).

Routing table entries installed by a user process may not specify the hash, reference count, use, or interface fields; these are filled in by the routing routines. If a route is in use when it is deleted (*rt_refcnt* is non-zero), the resources associated with it will not be reclaimed until all references to it are removed.

The routing code returns EEXIST if requested to duplicate an existing entry, ESRCH if requested to delete a non-existent entry, or ENOBUFS if insufficient resources were available to install a new route.

User processes read the routing tables through the **/dev/kmem** device.

The *rt_use* field contains the number of packets sent along the route. This value is used to select among multiple routes to the same destination. When multiple routes to the same destination exist, the least used route is selected.

A wildcard routing entry is specified with a zero destination address value. Wildcard routes are used only when the system fails to find a route to the destination host and network. The combination of wildcard routes and routing redirects can provide an economical mechanism for routing traffic.

FILES

/dev/kmem

SEE ALSO

ioctl(2), **route**(8C), **routed**(8C)

NAME

sd – driver for SCSI disk devices

CONFIG — SUN-3, SUN-3x, and SUN-4 SYSTEMS

controller si0 at vme24d16 ? csr 0x200000 priority 2 vector siintr 0x40
controller si0 at obio ? csr 0x140000 priority 2
disk sd0 at si0 drive 0 flags 0
disk sd1 at si0 drive 1 flags 0
disk sd2 at si0 drive 8 flags 0
disk sd3 at si0 drive 9 flags 0
disk sd4 at si0 drive 16 flags 0
disk sd6 at si0 drive 24 flags 0

controller sc0 at vme24d16 ? csr 0x200000 priority 2 vector scintr 0x40
disk sd0 at sc0 drive 0 flags 0
disk sd1 at sc0 drive 1 flags 0
disk sd2 at sc0 drive 8 flags 0
disk sd3 at sc0 drive 9 flags 0
disk sd4 at sc0 drive 16 flags 0
disk sd6 at sc0 drive 24 flags 0

The first two **controller** lines above specify the first and second SCSI host adapters for Sun-3, Sun-3x, and Sun-4 VME systems. The third **controller** line specifies the first and only SCSI host adapter on Sun-3/50 and Sun-3/60 systems.

The four lines following the **controller** specification lines define the available **disk** devices, **sd0 – sd6**.

The **flags** field is used to specify the SCSI device type to the host adapter. **flags** must be set to 0 to identify disk devices.

The **drive** value is calculated using the formula:

$8 * target + lun$

where *target* is the SCSI target, and *lun* is the SCSI logical unit number.

The next configuration block, following **si0** and **si1** above, describes the configuration for the older **sc0** host adapter. It uses the same configuration description as the **si0** host adapter.

CONFIG — SPARCsystem 330 and SUN-3/80 SYSTEMS

controller sm0 at obio ? csr 0xfa000000 priority 2
disk sd0 at sm0 drive 0 flags 0
disk sd1 at sm0 drive 1 flags 0
disk sd2 at sm0 drive 8 flags 0
disk sd3 at sm0 drive 9 flags 0
disk sd4 at sm0 drive 16 flags 0
disk sd6 at sm0 drive 24 flags 0

The SPARCsystem 330 and Sun-3/80 use an on-board SCSI host adapter, **sm0**. It follows the same rules as described above for the Sun-3, Sun-3x, and Sun-4 section.

CONFIG — SUN-4/110 SYSTEM

controller sw0 at obio 2 csr 0xa000000 priority 2
disk sd0 at sw0 drive 0 flags 0
disk sd1 at sw0 drive 1 flags 0
disk sd2 at sw0 drive 8 flags 0
disk sd3 at sw0 drive 9 flags 0
disk sd4 at sw0 drive 16 flags 0
disk sd6 at sw0 drive 24 flags 0

The Sun-4/110 uses an on-board SCSI host adapter, **sw0**. It follows the same rules as described above for the Sun-3, and Sun-4 section.

CONFIG — SUN-3/E SYSTEM

controller se0 at vme24d16 ? csr 0x300000 priority 2 vector se_intr 0x40
disk sd0 at se0 drive 0 flags 0
disk sd1 at se0 drive 1 flags 0
disk sd2 at se0 drive 8 flags 0
disk sd3 at se0 drive 9 flags 0

The Sun-3/E uses a VME-based SCSI host adapter, **se0**. It follows the same rules as described above for the Sun-3 and Sun-4 section.

CONFIG — Sun386i

controller wds0 at obmem ? csr 0xFB000000 dmachan 7 irq 16 priority 2
disk sd0 at wds0 drive 0 flags 0
disk sd1 at wds0 drive 8 flags 0
disk sd2 at wds0 drive 16 flags 0

The Sun386i configuration follows the same rules described above under the Sun-3 and Sun-4 configuration section. configuration section.

CONFIG — SPARCstation 1 SYSTEMS

device-driver esp
scsibus0 at esp
disk sd0 at scsibus0 target 3 lun 0
disk sd1 at scsibus0 target 1 lun 0
disk sd2 at scsibus0 target 2 lun 0
disk sd3 at scsibus0 target 0 lun 0

The SPARCstation 1 configuration files specify a device driver (**esp**), and a SCSI bus attached to that device driver, and then disks on that SCSI bus at the SCSI Target and Logical Unit addresses are specified.

DESCRIPTION

Files with minor device numbers 0 through 7 refer to various portions of drive 0. The standard device names begin with "sd" followed by the drive number and then a letter a-h for partitions 0-7 respectively. The character ? stands here for a drive number in the range 0-6.

The block-files access the disk using the system's normal buffering mechanism and are read and written without regard to physical disk records. There is also a "raw" interface that provides for direct transmission between the disk and the user's read or write buffer. A single read or write call usually results in one I/O operation; raw I/O is therefore considerably more efficient when many bytes are transmitted. The names of the raw files conventionally begin with an extra '**r**.'

I/O requests (such as **lseek** (2V)) to the SCSI disk must have an offset that is a multiple of 512 bytes (DEV_BSIZE), or the driver returns an EINVAL error. If the transfer length is not a multiple of 512 bytes, the transfer count is rounded up by the driver.

Disk Support

This driver handles the Adaptec ACB-4000 disk controller for ST-506 drives, the Emulex MD21 disk controller for ESDI drives, and embedded, CCS-compatible SCSI disk drives.

On Sun386i and SPARCstation 1 systems, this driver supports the CDC Wren III half-height, and Wren IV full-height SCSI disk drives.

The type of disk drive is determined using the SCSI inquiry command and reading the volume label stored on block 0 of the drive. The volume label describes the disk geometry and partitioning; it must be present or the disk cannot be mounted by the system.

The sd?a partition is normally used for the root file system on a disk, the sd?b partition as a paging area (e.g. swap), and the sd?c partition for pack-pack copying. sd?c normally maps the entire disk and may also be used as the mount point for secondary disks in the system. The rest of the disk is normally the sd?g partition. For the primary disk, the user file system is located here.

FILES

/dev/sd[0-6][a-h]	block files
/dev/rsd[0-6][a-h]	raw files

SEE ALSO

dkio(4S), **directory**(3V), **lseek**(2V), **read**(2V), **write**(2V)

Product Specification for Wren IV SCSI Model 94171
Product Specification for Wren III SCSI Model 94161
Product Specification for Wren III SCSI Model 94211
Emulex MD21 Disk Controller Programmer Reference Manual
Adaptec ACB-4000 Disk Controller OEM Manual

DIAGNOSTICS

sd?: sdtimer: I/O request timeout
A tape I/O operation has taken too long to complete. A device or host adapter failure may have occurred.

sd?: sdtimer: can't abort request
The driver is unable to find the request in the disconnect queue to notify the device driver that it has failed.

sd?: no space for inquiry data
sd?: no space for disk label
The driver was unable to get enough space for temporary storage. The driver is unable to open the disk device.

sd?: <%s>
The driver has found a SCSI disk device and opened it for the first time. The disk label is displayed to notify the user.

sd?: SCSI bus failure
A host adapter error was detected. The system may need to be rebooted.

sd?: single sector I/O failed
The driver attempted to recover from a transfer by writing each sector, one at a time, and failed. The disk needs to be reformatted to map out the new defect causing this error.

sd?: retry failed
sd?: rezero failed
A disk operation failed. The driver first tries to recover by retrying the command, if that fails, the driver rezeros the heads to cylinder 0 and repeats the retries. A failure of either the retry or rezero operations results in these warning messages; the error recovery operation continues until the retry count is exhausted. At that time a hard error is posted.

sd?: request sense failed
The driver was attempting to determine the cause of an I/O failure and was unable to get more information. This implies that the disk device may have failed.

sd?: warning, abs. block %d has failed %d times
The driver is warning the user that the specified block has failed repeatedly.

sd?: block %d needs mapping
sd?: reassigning defective abs. block %d
The specified block has failed repeatedly and may soon become an unrecoverable failure. If the driver does not map out the specified block automatically, it is recommend that the user correct the problem.

sd?: reassign block failed
The driver attempted to map out a block having excessive soft errors and failed. The user needs to run format and repair the disk.

sd?%c: cmd how blk %d (rel. blk %d)
sense key(0x%x): %s, error code(0x%x): %s
An I/O operation (**cmd**), encountered an error condition at absolute block (**blk %d**), partition (**sd?%c:**), or relative block (**rel. block%d**). The error recovery operation (how) indicates whether it *retry*'ed, *restored*, or *failed*. The **sense key** and **error code** of the error are displayed for diagnostic purposes. The absolute **blk** of the the error is used for mapping out the defective block. The **rel. blk** is the block (sector) in error, relative to the beginning of the partition involved. This is useful for using **icheck**(8) to repair a damaged file structure on the disk.

SPARCstation 1 Diagnostics

The diagnostics for SPARCstation 1 are much like as above. Below are some additional diagnostics you might see on a SPARCstation 1:

sd?: SCSI transport failed: reason 'xxxx': {retrying|giving up}
The host adapter has failed to transport a command to the target for the reason stated. The driver will either retry the command or, ultimately, give up.

sd?: disk not responding to selection
The target disk isn't responding. You may have accidently kicked a power cord loose.

sd?: disk ok
The target disk is now responding again.

sd?: disk offline
The driver has decided that the target disk is no longer there.

BUGS

These disk drivers assume that you don't have removable media drives, and also that in order to operate normally, a valid Sun disk label must be in sector zero.

A logical block size of 512 bytes is assumed (and enforced on SPARCstation 1).

NAME

sockio – ioctls that operate directly on sockets

SYNOPSIS

#include <sys/sockio.h>

DESCRIPTION

The IOCTL's listed in this manual page apply directly to sockets, independent of any underlying protocol. Note: the **setsockopt** system call (see **getsockopt**(2)) is the primary method for operating on sockets as such, rather than on the underlying protocol or network interface. **ioctl**s for a specific network interface or protocol are documented in the manual page for that interface or protocol.

SIOCSPGRP The argument is a pointer to an **int**. Set the process-group ID that will subsequently receive **SIGIO** or **SIGURG** signals for the socket referred to by the descriptor passed to **ioctl** to the value of that **int**.

SIOCGPGRP The argument is a pointer to an **int**. Set the value of that **int** to the process-group ID that is receiving **SIGIO** or **SIGURG** signals for the socket referred to by the descriptor passed to **ioctl**.

SIOCCATMARK The argument is a pointer to an **int**. Set the value of that **int** to 1 if the read pointer for the socket referred to by the descriptor passed to **ioctl** points to a mark in the data stream for an out-of-band message, and to 0 if it does not point to a mark.

SEE ALSO

ioctl(2), **getsockopt**(2), **filio**(4)

NAME

sr – driver for CDROM SCSI controller

CONFIG — SPARCstation 1 and SPARCserver

disk sr0 at scsibus0 target 6 lun 0

CONFIG — SUN-4/330 SYSTEMS

disk sr0 at sm0 drive 060 flags 2

CONFIG — SUN-4 SYSTEMS

disk sr0 at sc0 drive 060 flags 2
disk sr0 at si0 drive 060 flags 2

AVAILABILITY

SPARCstation 1, SPARCserver 1, and Sun-4/330 systems only.

DESCRIPTION

CDROM is a removable read-only direct-access device connected to the system's SCSI bus. CDROM drives are designed to work with any disc that meets the Sony-Philips "red-book" or "yellow-book" documents. They can read CDROM data discs, digital audio discs (Audio CD's) or combined-mode discs (that is, some tracks are audio, some tracks are data). A CDROM disc is singled sided containing approximately 540 mega-bytes of data or 74 minutes of audio.

The CDROM drive controller is set up as SCSI target 6. There is only a single logically unit number 0. Therefore, the minor device number is always 0.

Since all the other SCSI target ids has been reserved by the system, the system only supports one CDROM drive. The device names are **/dev/sr0** for block device and **/dev/rsr0** for character device.

The device driver supports **open**(2V), **read**(2V), **close**(2V) function calls through its block device and character device interface. In addition, it supports ioctl function call through the character device interface. When the device is first opened, the CDROM drive's eject button will be disabled (which prevents the manual removal of the disc) until the last **close**(2V) is called.

CDROM Drive Support

This driver supports the SONY CDU-8012 CDROM drive controller and other CDROM drives which has the same SCSI command set as the SONY CDU-8012. The type of CDROM drive is determined using the SCSI inquiry command.

There is no volume label stored on the CDROM. The disc geometry and paritioning information is always the same. If the CDROM is in ISO 9660 or **High Sierra Disk** format, it can be mounted as a file system.

FILES

/dev/sr0	block files
/dev/rsr0	raw files

SEE ALSO

cdromio(4S), **fstab**(5), **mount**(8)

NAME

st – driver for SCSI tape devices

CONFIG — SUN-3, SUN-3x, SUN-4 SYSTEMS

controller si0 at vme24d16 ? csr 0x200000 priority 2 vector siintr 0x40
controller si1 at vme24d16 ? csr 0x204000 priority 2 vector siintr 0x41
controller si0 at obio ? csr 0x140000 priority 2
tape st0 at si0 drive 32 flags 1
tape st1 at si0 drive 40 flags 1
tape st2 at si1 drive 32 flags 1
tape st3 at si1 drive 40 flags 1

controller sc0 at vme24d16 ? csr 0x200000 priority 2 vector scintr 0x40
tape st0 at sc0 drive 32 flags 1
tape st1 at sc0 drive 40 flags 1

The first two **controller** lines above specify the first and second SCSI host adapters for Sun-3, Sun-3x, and Sun-4 VME systems. The third **controller** line specifies the first and only SCSI host adapter on Sun-3/50 and Sun-3/60 systems.

Following the **controller** specification lines are four lines which define the available **tape** devices, **st0–st3**. The first two **tape** devices, **st0** and **st1**, are on the first **controller**, **si0**. The next two **tape** devices, **st2** and **st3**, are on the second **controller**, **si1**.

The **flags** field is used to specify the SCSI device type to the host adapter. The **flags** field must be set to 1 to identify **tape** devices.

The **drive** value is calculated using the formula:

$8 * target + lun$

where *target* is the SCSI target, and *lun* is the SCSI logical unit number.

The next configuration block, following **si0** and **si1** above, describes the older **sc0** host adapter configuration. It follows the same configuration description as the **si0** host adapter.

CONFIG — SPARCsystem 330, SUN–3/80 SYSTEMS

controller sm0 at obio ? csr 0xfa000000 priority 2
tape st0 at sm0 drive 32 flags 1
tape st1 at sm0 drive 40 flags 1

The SPARCsystem 330 and Sun-3/80 use an on-board SCSI host adapter, **sm0**, which follows the rules described above in the Sun-3, Sun-3x, and Sun-4 section.

CONFIG — SUN–4/110 SYSTEM

controller sw0 at obio 2 csr 0xa000000 priority 2
tape st0 at sw0 drive 32 flags 1
tape st1 at sw0 drive 40 flags 1

The Sun-4/110 uses an on-board SCSI host adapter, **sw0**, which follows the rules described above in the Sun-3, Sun-3x, and Sun-4 section.

CONFIG — SUN–3/E SYSTEM

controller se0 at vme24d16 ? csr 0x300000 priority 2 vector se_intr 0x40
tape st0 at se0 drive 32 flags 1
tape st1 at se0 drive 40 flags 1

The Sun-3/E uses a VME-based SCSI host adapter, **se0**, which follows the rules described above for Sun-3, Sun-3x, and Sun-4 systems.

CONFIG — Sun386i

controller wds0 at obmem ? csr 0xFB000000 dmachan 7 irq 16 priority 2
tape st0 at wds0 drive 32 flags 1

The Sun386*i* configuration follows the rules described above in the Sun-3, Sun-3x, and Sun-4 configuration section.

CONFIG — SPARCstation 1 SYSTEM

device-driver esp
scsibus0 at esp
tape st0 at scsibus0 target 4 lun 0
tape st1 at scsibus0 target 5 lun 1

The SPARCstation 1 configuration files specify a device driver (**esp**), and a SCSI bus attached to that device driver, and then tapes on that SCSI bus at the SCSI Target and Logical Unit addresses are specified.

DESCRIPTION

The **st** device driver is an interface to various SCSI tape devices. Supported 1/4–inch cartridge devices include the Archive Viper QIC–150 streaming tape drive, the Emulex MT–02 tape controller, and the Sysgen SC4000 (except on SPARCstation 1) tape controller. **st** provides a standard interface to these various devices, see **mtio**(4) for details.

The driver can be opened with either rewind on close (**/dev/rst***) or no rewind on close (**/dev/nrst***) options. A maximum of four tape formats per device are supported (see FILES below). The tape format is specified using the device name. The four rewind on close formats for **st0**, for example, are **/dev/rst0**, **/dev/rst8**, **/dev/rst16**, and **/dev/rst24**.

Read Operation

Fixed-length I/O tape devices require the number of bytes read or written to be a multiple of the physical record size. For example, 1/4–inch cartridge tape devices only read or write multiples of 512 bytes.

Fixed-length tape devices read or write multiple records if the blocking factor is greater than 64512 bytes (minphys limit). These multiple writes are limited to 64512 bytes. For example, if a write request is issued for 65536 bytes using a 1/4–inch cartridge tape, two writes are issued; the first for 64512 bytes and the second for 1024 bytes.

Tape devices, which support variable-length I/O operations, such as 1/2–inch reel tape, may read or write a range of 1 to 65535 bytes. If the record size exceeds 65535 bytes, the driver reads or writes multiple records to satisfy the request. These multiple records are limited to 65534 bytes. As an example, if a write request for 65540 bytes is issued using 1/2–inch reel tape, two records are written; one for 65534 bytes followed by one for 6 bytes.

If the driver is opened for reading in a different format than the tape is written in, the driver overrides the user selected format. For example, if a 1/4–inch cartridge tape is written in QIC-24 format and opened for reading in QIC–11, the driver will detect a read failure on the first read and automatically switch to QIC–24 to recover the data.

Note: If the **/dev/*st[0–3]** format is used, no indication is given that the driver has overridden the user selected format. Other formats issue a warning message to inform the user of an overridden format selection. Some devices automatically perform this function and do not require driver support (1/2–inch reel and QIC–150 tape drives for example).

If a file mark is encountered during reading, no error is reported but the number of bytes transferred is zero. The next read operation reads into the next file.

End of media is indicated by two successive zero transfer counts. No further reading should be performed past the end of recorded media.

If the read request size is 2048 bytes, the tape driver behaves as a disk device and honors seek positioning requests (see **lseek**(2)). If a file mark is crossed during a read operation, this function is disabled.

Write Operation

Writing is allowed at either the beginning of tape or after the last written file on the tape. Writing from the beginning of tape is performed in the user-specified format. The original tape format is used for appending onto previously written tapes. A warning message is issued if the driver has to override the user-specified format.

Care should be used when appending files onto 1/2–inch reel tape devices, since an extra file mark is appended after the last file to mark the end of recorded media. In other words, the last file on the tape ends with two file marks instead of one. This extra file mark must be overwritten to prevent the creation of a null file. To facilitate write append operations, a space to the end of recorded media **ioctl()** is provided to eliminate this problem by having the driver perform the positioning operation.

If the end of tape is encountered during writing, no error is reported but the number of bytes transferred is zero and no further writing is allowed. Trailer records may be written by first writing a file mark followed by the trailer records. It is important that these trailer records be kept as short as possible to prevent data loss.

Close Operation

If data was written, a file mark is automatically written by the driver upon close. If the rewinding device name is used, the tape will be rewound after the file mark is written. If the user wrote a file mark prior to closing, then no file mark is written upon close. If a file positioning **ioctl()**, like rewind, is issued after writing, a file mark is written before repositioning the tape.

Note: For 1/2–inch reel tape devices, two file marks are written to mark the end of recorded media before rewinding or performing a file positioning **ioctl()**.**If**the**user**wrote mark before closing a 1/2–inch reel tape device, the driver will always write a file mark before closing to insure that the end of recorded media is marked properly.

If no data was written and the driver was opened for WRITE-ONLY access, a file mark is written thus creating a null file.

IOCTLS

The following ioctls are supported: forwardspace record, forwardspace file, backspace record, backspace file, backspace file mark, rewind, write file mark, offline, erase, retension, space to EOM, and get status.

The backspace file and forwardspace file tape operations are inverses. Thus, a forwardspace "–1" file is equivalent to a backspace "1" file. A backspace "0" file is the same as forwardspace "0" file; both position the tape device to the beginning of the current file.

Backspace file mark moves the tape backwards by file marks. The tape position will end on the beginning of tape side of the desired file mark. Devices which do not support this function, such as 1/4–inch cartridge tape, return an ENXIO error.

Backspace record and forwardspace record operations perform much like space file operations, except that they move by records instead of files. Variable-length I/O devices (1/2–inch reel, for example) space actual records; fixed-length I/O devices space physical records (blocks). 1/4–inch cartridge tape, for example, spaces 512 byte physical records. The status ioctl residue count contains the number of files or records not skipped. Record skipping does not go past a file mark; file skipping does not go past the end of recorded media.

Spacing to the end of recorded media positions the tape at a location just after the last file written on the tape. For 1/4–inch cartridge tape, this is after the last file mark on the tape. For 1/2–inch reel tape, this is just after the first file mark but before the second (and last) file mark on the tape. Additional files can then be appended onto the tape from that point.

The offline ioctl rewinds and, if appropriate, takes the device offline by unloading the tape. Tape must be inserted before the tape device can be used again.

The erase ioctl rewinds the tape, erases it completely, and returns to the beginning of tape.

The retension ioctl only applies to 1/4–inch cartridge tape devices. It is used to restore tape tension improving the tape's soft error rate after extensive start-stop operations or long-term storage. Devices which do not support this function, such as 1/2–inch reel tape, return an ENXIO error.

The get status ioctl call returns the drive id (mt_type), sense key error (mt_erreg), file number (mt_fileno), and record number (mt_blkno) of the last error. The residue count (mt_resid) is set to the number of bytes not transferred or files/records not spaced.

Note: The error status is reset by the get status ioctl call or the next read, write, or other ioctl operation. If no error has occurred (sense key is zero), the current file and record position are returned.

ERRORS

EACCES The driver is opened for write access and the tape is write protected, or an attempt is made to write on a write protected tape. For writing with QIC–150 tape drives, this error is also reported if the wrong tape media is used for writing.

EBUSY The tape device is already in use.

EIO During opening, the tape device is not ready because either no tape is in the drive, or the drive is not on-line. Once open, this error is returned if the requested I/O transfer could not be completed.

EINVAL The number of bytes read or written is not a multiple of the physical record size (fixed-length tape devices only).

ENXIO During opening, the tape device does not exist. On ioctl functions, this indicates that the tape device does not support the ioctl function.

FILES

For QIC–150 tape devices (Archive Viper):

/dev/rst[0–3] QIC–150 Format
/dev/rst[8–11] QIC–150 Format
/dev/rst[16–20] QIC–150 Format
/dev/rst[24–28] QIC–150 Format
/dev/nrst[0–3] non-rewinding QIC–150 Format
/dev/nrst[8–11] non-rewinding QIC–150 Format
/dev/nrst[16–19] non-rewinding QIC–150 Format
/dev/nrst[24–27] non-rewinding QIC–150 Format

For QIC–24 tape devices (Emulex MT–02 and Sysgen SC4000):

/dev/rst[0–3] QIC–11 Format
/dev/rst[8–11] QIC–24 Format
/dev/rst[16–20] QIC–24 Format
/dev/rst[24–28] QIC–24 Format
/dev/nrst[0–3] non-rewinding QIC–11 Format
/dev/nrst[8–11] non-rewinding QIC–24 Format
/dev/nrst[16–19] non-rewinding QIC–24 Format
/dev/nrst[24–27] non-rewinding QIC–24 Format

Note: The QIC–24 format is preferred over QIC–11 for Sun-3, Sun-3x, Sun-4, and Sun386*i* systems.

SEE ALSO

mt(1), **tar**(1), **mtio**(4), **dump**(8), **restore**(8)

Archive Viper QIC–150 Tape Drive Product Specification
Emulex MT–02 Intelligent Tape Controller Product Specification
Sysgen SC4000 Intelligent Tape Controller Product Specification

DIAGNOSTICS

st?: sttimer: I/O request timeout
A tape I/O operation has taken too long to complete. A device or host adapter failure may have occurred.

st?: sttimer: can't abort request
The driver is unable to find the request in the disconnect que to notify the device driver that it has failed. A SCSI bus reset is issued to recover from this error.

st?: unknown SCSI device found
The SCSI device is not a tape device; it is some other type of SCSI device.

st?: warning, unknown tape drive found
The driver does not recognize the tape device. Only the default tape density is used; block size is set to the value specified by the tape drive.

st?: tape is write protected
The tape is write protected.

st?: wrong tape media for writing
For QIC–150 tape drives, this indicates that the user is trying to write on a DC–300XL (or equivalent) tape. Only DC–6150 (or equivalent) tapes can be used for writing.
Note: DC–6150 was formerly known as DC–600XTD.

st?: warning, rewinding tape
The driver is rewinding tape in order to set the tape format.

st?: warning, using alternate tape format
The driver is overriding the user-selected tape format and using the previously used format.

st?: warning, tape rewound
For Sysgen tape controllers, the tape may be rewound as a result of getting sense data.

st?: format change failed
The tape drive rejected the mode select command to change the tape format.

st?: file mark write failed
The driver was unable to write a file mark.

st?: warning, The tape may be wearing out or the head may need cleaning.
st?: read retries= %d, file= %d, block= %d
st?: write retries= %d, file= %d, block= %d
The number of allowable soft errors has been exceeded for this tape. Either the tape heads need cleaning or the tape is wearing out. If the tape is wearing out, continued usage of it is not recommended.

st?: illegal command
The SCSI command just issued was illegal. This message can result from issuing an inappropriate command, such as trying to write over previously written files on the tape. On foreign tape devices, this can also be caused by selecting the wrong tape format.

st?: error: sense key(0x%x): %s, error code(0x%x): %s
An error has occurred. The sense key message and error code are displayed for diagnostic purposes.

st?: stread: not modulo %d block size
st?: stwrite: not modulo %d block size
The read or write request size must be a multiple of the %d physical block size.

st?: file positioning error
st?: block positioning error
The driver was unable to position the tape to the desired file or block (record). This is probably caused by a damaged tape.

st?: SCSI transport failed: reason 'xxxx': {retrying|giving up}

The host adapter has failed to transport a command to the target for the reason stated. The driver will either retry the command or, ultimately, give up (SPARCstation 1) only.

BUGS

Foreign tape devices which do not return a BUSY status during tape loading prevent user commands from being held until the device is ready. The user must delay issuing any tape operations until the tape device is ready. This is not a problem for Sun supplied tape devices.

Foreign tape devices which do not report a blank check error at the end of recorded media cause file positioning operations to fail. Some tape drives for example, mistakenly report media error instead of blank check error.

"Cooked" mode for read and write operations is not supported.

Systems using the older **sc0** host adapter or the Sysgen SC4000 tape controller, prevent disk I/O over the SCSI bus while the tape is in use (during a rewind for example). This problem is caused by the fact that they do not support disconnect/reconnect to free the SCSI bus. Newer tape devices, like the the Emulex MT–02, and host adapters, like **si0**, eliminate this problem.

Some older systems may not support the QIC–24 format, and may complain (or exhibit erratic behavior) when the user attempts to use this format.

SPARCstation 1 does not support the Sysgen SC4000 tape controller, nor does it support 1/2" variable record length operations, record space operations, or implied seeking.

NAME

streamio – STREAMS ioctl commands

SYNOPSIS

#include <stropts.h>
int ioctl (fd, command, arg)
int fd, command;

DESCRIPTION

STREAMS (see **intro**(2)) ioctl commands are a subset of **ioctl**(2) commands that perform a variety of control functions on STREAMS. The arguments *command* and *arg* are passed to the file designated by *fd* and are interpreted by the *stream*head. Certain combinations of these arguments may be passed to a module or driver in the stream.

fd 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.

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 *fd* 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. Subsequent system calls will fail with *errno* set to this value.

IOCTLS

The following **ioctl** commands, with error values indicated, are applicable to all STREAMS files:

I_PUSH Pushes the module whose name is pointed to by *arg* onto the top of the current stream, just below the *stream*head. It then calls the open routine of the newly-pushed module.

I_PUSH will fail if one of the following occurs:

EINVAL	The module name is invalid.
EFAULT	*arg* points outside the allocated address space.
ENXIO	The open routine of the new module failed.
ENXIO	A hangup is received on the stream referred to by *fd*.

I_POP Removes the module just below the *stream head* of the stream pointed to by *fd*. *arg* should be 0 in an **I_POP** request.

I_POP will fail if one of the following occurs:

EINVAL	No module is present on *stream*.
ENXIO	A hangup is received on the stream referred to by *fd*.

I_LOOK Retrieves the name of the module just below the *stream head* of the stream pointed to by *fd*, and places it in a null-terminated character string pointed at by *arg*. The buffer pointed to by *arg* should be at least **FMNAMESZ+1** bytes long. An '**#include <sys/conf.h>**' declaration is required.

I_LOOK will fail if one of the following occurs:

EFAULT	*arg* points outside the allocated address space of the process.
EINVAL	No module is present on *stream*.

I_FLUSH This request flushes all input and/or output queues, depending on the value of *arg*. Legal *arg* values are:

FLUSHR	Flush read queues.
FLUSHW	Flush write queues.
FLUSHRW	Flush read and write queues.

I_FLUSH will fail if one of the following occurs:

EAGAIN	No buffers could be allocated for the flush message.
EINVAL	The value of *arg* is invalid.
ENXIO	A hangup is received on the stream referred to by *fd*.

I_SETSIG Informs the *stream head* that the user wishes the kernel to issue the **SIGPOLL** signal (see **sigvec**(2)) when a particular event has occurred on the stream associated with *fd*. **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 of the following constants:

S_INPUT	A non-priority message has arrived on a *stream head* read queue, and no other messages existed on that queue before this message was placed there. This is set even if the message is of zero length.
S_HIPRI	A priority message is present on the *stream head* read queue. This is set even if the message is of zero length.
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_MSG	A STREAMS signal message that contains the **SIGPOLL** signal has reached the front of the *stream head* read queue.

A user process may choose to be signaled only of priority messages by setting the *arg* bitmask to the value **S_HIPRI**.

Processes that wish to receive **SIGPOLL** signals must explicitly register to receive them using **I_SETSIG**. If several processes register to receive this 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_SETSIG will fail if one of the following occurs:

EINVAL	The value of *arg* is invalid or *arg* is zero and the process is not registered to receive the **SIGPOLL** signal.
EAGAIN	A data structure could not be allocated to store the signal request.

I_GETSIG Returns the events for which the calling process is currently registered to be sent a **SIGPOLL** signal. The events are returned as a bitmask pointed to by *arg*, where the events are those specified in the description of **I_SETSIG** above.

I_GETSIG will fail if one of the following occurs:

EINVAL	The process is not registered to receive the **SIGPOLL** signal.
EFAULT	*arg* points outside the allocated address space of the process.

I_FIND This request compares the names of all modules currently present in the stream to the name pointed to by *arg*, and returns 1 if the named module is present in the stream. It returns 0 if the named module is not present.

I_FIND will fail if one of the following occurs:

EFAULT	*arg* points outside the allocated address space of the process.
EINVAL	*arg* does not point to a valid module name.

I_PEEK This request allows a user to retrieve the information in the first message on the *stream head* read queue without taking the message off the queue. *arg* points to a *strpeek* structure which contains the following members:

```
struct strbuf   ctlbuf;
struct strbuf   databuf;
long            flags;
```

The *maxlen* field in the *ctlbuf* and *databuf strbuf* structures (see **getmsg**(2)) must be set to the number of bytes of control information and/or data information, respectively, to retrieve. If the user sets *flags* to **RS_HIPRI**, **I_PEEK** will only look for a priority message on the *stream head* read queue.

I_PEEK returns 1 if a message was retrieved, and returns 0 if no message was found on the *stream head* read queue, or if the **RS_HIPRI** flag was set in *flags* and a priority message was not present on the *stream head* read queue. It does not wait for a message to arrive. On return, *ctlbuf* specifies information in the control buffer, *databuf* specifies information in the data buffer, and *flags* contains the value 0 or **RS_HIPRI**.

I_PEEK will fail if one of the following occurs:

EFAULT	*arg* points, or the buffer area specified in *ctlbuf* or *databuf* is, outside the allocated address space of the process.

I_SRDOPT Sets the read mode using the value of the argument *arg*. Legal *arg* values are:

RNORM	Byte-stream mode, the default.
RMSGD	Message-discard mode.
RMSGN	Message-nondiscard mode.

Read modes are described in **read**(2V).

I_SRDOPT will fail if one of the following occurs:

EINVAL	*arg* is not one of the above legal values.

I_GRDOPT Returns the current read mode setting in an *int* pointed to by the argument *arg*. Read modes are described in **read**(2V).

I_GRDOPT will fail if one of the following occurs:

EFAULT	*arg* points outside the allocated address space of the process.

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_NREAD will fail if one of the following occurs:

EFAULT *arg* points outside the allocated address space of the process.

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             fd;
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. *fd* specifies the file descriptor of the other stream and *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 to the *fd* stream's driver read queue 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. A non-priority message is created if *flags* is set to 0, and a priority message is created if *flags* is set to **RS_HIPRI**. For non-priority messages, **I_FDINSERT** will block if the stream write queue is full due to internal flow control conditions. For priority messages, **I_FDINSERT** does not block on this condition. For non-priority messages, **I_FDINSERT** does not block when the write queue is full and **O_NDELAY** is set. Instead, it fails and sets *errno* to EAGAIN.

I_FDINSERT also blocks, unless prevented by lack of internal resources, waiting for the availability of message blocks in the *stream*, regardless of priority or whether **O_NDELAY** has been specified. No partial message is sent.

I_FDINSERT will fail if one of the following occurs:

EAGAIN A non-priority message was specified, the **O_NDELAY** flag is set, and the stream write queue is full due to internal flow control conditions.

EAGAIN Buffers could not be allocated for the message that was to be created.

EFAULT *arg* points, or the buffer area specified in *ctlbuf* or *databuf* is, outside the allocated address space of the process.

EINVAL — *fd* in the *strfdinsert* structure is not a valid, open stream file descriptor; the size of a pointer plus *offset* is greater than the *len* field for the buffer specified through *ctlptr*; *offset* does not specify a properly-aligned location in the data buffer; an undefined value is pointed to by *flags*.

ENXIO — A hangup is received on the stream referred to by *fd*.

ERANGE — The *len* field for the buffer specified through *databuf* does not fall within the range specified by the maximum and minimum packet sizes of the topmost stream module, or the *len* field for the buffer specified through *databuf* is larger than the maximum configured size of the data part of a message, or the *len* field for the buffer specified through *ctlbuf* is larger than the maximum configured size of the control part of a message.

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 permit a process to specify timeouts and variable-sized amounts of data when sending an **ioctl** request 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 interval for these requests is 15 seconds. The **O_NDELAY** (see **open**(2V)) flag has no effect on this call.

To send requests downstream, *arg* must point to a *strioctl* structure which contains the following members:

```
int     ic_cmd;     /* downstream command */
int     ic_timout;  /* ACK/NAK timeout */
int     ic_len;     /* length of data arg */
char    *ic_dp;     /* ptr to data arg */
```

ic_cmd is the internal ioctl command intended for a 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. *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 the driver in the stream can return).

The *stream head* will convert the information pointed to by the *strioctl* structure to an internal ioctl command message and send it downstream.

I_STR will fail if one of the following occurs:

EAGAIN — Buffers could not be allocated for the **ioctl** message.

EFAULT	*arg* points, or the buffer area specified by *ic_dp* and *ic_len* (separately for data sent and data returned) is, outside the allocated address space of the process.
EINVAL	*ic_len* is less than 0 or *ic_len* is larger than the maximum configured size of the data part of a message or *ic_timout* is less than −1.
ENXIO	A hangup is received on the stream referred to by *fd*.
ETIME	A downstream **ioctl** timed out before acknowledgement was received.

An **I_STR** can also fail while waiting for an acknowledgement if a message indicating an error or a hangup is received at the *stream*head. In addition, an error code can be returned in the positive or negative acknowledgement message, in the event the **ioctl** command sent downstream fails. For these cases, **I_STR** will fail with *errno* set to the value in the message.

I_SENDFD Requests the stream associated with *fd* 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 integer 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 (see **intro**(2)) of the *stream head* at the other end of the stream pipe to which it is connected.

I_SENDFD will fail if one of the following occurs:

EAGAIN	The sending stream is unable to allocate a message block to contain the file pointer.
EAGAIN	The read queue of the receiving *stream head* is full and cannot accept the message sent by **I_SENDFD**.
EBADF	*arg* is not a valid, open file descriptor.
EINVAL	*fd* is not connected to a stream pipe.
ENXIO	A hangup is received on the stream referred to by *fd*.

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 an *strrecvfd* data structure containing the following members:

```
int fd;
unsigned short uid;
unsigned short 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_NDELAY** is not set (see **open**(2V)), **I_RECVFD** will block until a message is present at the *stream*head. If **O_NDELAY** 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 an **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_RECVFD will fail if one of the following occurs:

EAGAIN	A message was not present at the *stream head* read queue, and the **O_NDELAY** flag is set.
EBADMSG	The message at the *stream head* read queue was not a message containing a passed file descriptor.
EFAULT	*arg* points outside the allocated address space of the process.
EMFILE	Too many descriptors are active.
ENXIO	A hangup is received on the stream referred to by *fd*.

The following four commands are used for connecting and disconnecting multiplexed STREAMS configurations.

I_LINK Connects two streams, where *fd* 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** causes 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 a −1 on failure.

I_LINK will fail if one of the following occurs:

ENXIO	A hangup is received on the stream referred to by *fd*.
ETIME	The **ioctl** timed out before an acknowledgement was received.
EAGAIN	Storage could not be allocated to perform the **I_LINK**.
EBADF	*arg* is not a valid, open file descriptor.
EINVAL	The stream referred to by *fd* does not support multiplexing.
EINVAL	*arg* is not a stream, or is already linked under a multiplexor.
EINVAL	The specified link operation would cause a "cycle" in the resulting configuration; that is, if a given *stream head* is linked into a multiplexing configuration in more than one place.

An **I_LINK** can also fail while waiting for the multiplexing driver to acknowledge the link request, if a message indicating an error or a hangup is received at the *stream head* of *fd*. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, **I_LINK** will fail with *errno* set to the value in the message.

I_UNLINK Disconnects the two streams specified by *fd* and *arg*. *fd* is the file descriptor of the stream connected to the multiplexing driver. *arg* is the multiplexor ID number that was returned by the **ioctl I_LINK** command when a stream was linked below the multiplexing driver. If *arg* is −1, then all streams which were linked to *fd* are disconnected. As in **I_LINK**, this command requires the multiplexing driver to acknowledge the unlink.

I_UNLINK will fail if one of the following occurs:

ENXIO	A hangup is received on the stream referred to by *fd*.

ETIME	The **ioctl** timed out before an acknowledgement was received.
EAGAIN	Buffers could not be allocated for the acknowledgement message.
EINVAL	The multiplexor ID number was invalid.

An I_UNLINK can also fail while waiting for the multiplexing driver to acknowledge the link request, if a message indicating an error or a hangup is received at the *stream head* of *fd*. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, I_UNLINK will fail with *errno* set to the value in the message.

SEE ALSO

close(2V), **fcntl**(2V), **getmsg**(2), **intro**(2), **ioctl**(2), **open**(2V), **poll**(2), **putmsg**(2), **read**(2V), **sigvec**(2), **write**(2V)

STREAMS Programmer's Guide
STREAMS Primer

NAME

taac – Sun applications accelerator

CONFIG

taac0 at vme32d32 ? csr 0x28000000

CONFIG – SUN-3/SUN-4 SYSTEMS

device taac0 at vme32d32 1 csr 0x28000000
device taac0 at vme32d32 2 csr 0xf8000000
device taac0 at vme32d32 3 csr 0x28000000

The first line should be used to generate a kernel for Sun-3/160, Sun-3/260, Sun-4/260, Sun-4/370 and Sun-4/460 systems. The second line should be used to generate a kernel for Sun-4/110 systems; and the last line should be used to generate a kernel for Sun-4/330 systems.

CONFIG – SUN-4/150 SYSTEMS

device taac0 at vme32d32 2 csr 0xf8000000

AVAILABILITY

TAAC-1 can only be used in Sun VME-bus packages with 4 or more full size (9U) slots.

DESCRIPTION

The taac interface supports the optional TAAC-1 Applications Accelerator. This add-on device is composed of a very-long-instruction-word computation engine, coupled with an 8MB memory array. This memory area can be used as a frame buffer or as storage for large data sets.

the Sun-4/150 VME address space is limited to 28 bits. The TAAC-1 must be reconfigured to work in this package. See *Configuration Procedures fro the TAAC-1 Application Accelerator Board Set.*

Programs can be downloaded for execution on the TAAC-1 directly, they can be executed by the host processor, or the host processor and the TAAC-1 engine can be used in combination. See the *TAAC-1 User's Guide* for detailed information on accessing the TAAC-1 from the host. This manual also describes the C compiler, the programming tools, and the support libraries for the TAAC-1.

Programs on the host processor gain access to the TAAC-1 registers and memory by using **mmap**(2).

SEE ALSO

mmap(2)

TAAC-1 Application Accelerator: User Guide
Configuration Procedures for the TAAC-1 Application Accelerator Board Set

NAME

tcp – Internet Transmission Control Protocol

SYNOPSIS

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

s = socket(AF_INET, SOCK_STREAM, 0);

DESCRIPTION

TCP is the virtual circuit protocol of the Internet protocol family. It provides reliable, flow-controlled, in order, two-way transmission of data. It is a byte-stream protocol used to support the SOCK_STREAM abstraction. TCP is layered above the Internet Protocol (IP), the Internet protocol family's unreliable internetwork datagram delivery protocol.

TCP uses IP's host-level addressing and adds its own per-host collection of "port addresses". The endpoints of a TCP connection are identified by the combination of an IP address and a TCP port number. Although other protocols, such as the User Datagram Protocol (UDP), may use the same host and port address format, the port space of these protocols is distinct. See **inet**(4F) for details on the common aspects of addressing in the Internet protocol family.

Sockets utilizing TCP are either "active" or "passive". Active sockets initiate connections to passive sockets. Both types of sockets must have their local IP address and TCP port number bound with the **bind**(2) system call after the socket is created. By default, TCP sockets are active. A passive socket is created by calling the **listen**(2) system call after binding the socket with **bind** . This establishes a queueing parameter for the passive socket. After this, connections to the passive socket can be received with the **accept**(2) system call. Active sockets use the **connect**(2) call after binding to initiate connections.

By using the special value INADDR_ANY, the local IP address can be left unspecified in the **bind** call by either active or passive TCP sockets. This feature is usually used if the local address is either unknown or irrelevant. If left unspecified, the local IP address will be bound at connection time to the address of the network interface used to service the connection.

Once a connection has been established, data can be exchanged using the **read**(2V) and **write**(2V) system calls.

TCP supports one socket option which is set with **setsockopt** and tested with **getsockopt**(2). Under most circumstances, TCP sends data when it is presented. When outstanding data has not yet been acknowledged, it gathers small amounts of output to be sent in a single packet once an acknowledgement is received. For a small number of clients, such as window systems that send a stream of mouse events which receive no replies, this packetization may cause significant delays. Therefore, TCP provides a boolean option, TCP_NODELAY (defined in **<netinet/tcp.h>**), to defeat this algorithm. The option level for the **setsockopt** call is the protocol number for TCP, available from **getprotobyname** (see **getprotoent**(3N)).

Options at the IP level may be used with TCP; see **ip**(4P).

TCP provides an urgent data mechanism, which may be invoked using the out-of-band provisions of **send**(2). The caller may mark one byte as "urgent" with the MSG_OOB flag to **send**(2). This causes an "urgent pointer" pointing to this byte to be set in the TCP stream. The receiver on the other side of the stream is notified of the urgent data by a SIGURG signal. The SIOCATMARK ioctl returns a value indicating whether the stream is at the urgent mark. Because the system never returns data across the urgent mark in a single **read**(2V) call, it is possible to advance to the urgent data in a simple loop which reads data, testing the socket with the SIOCATMARK ioctl, until it reaches the mark.

Incoming connection requests that include an IP source route option are noted, and the reverse source route is used in responding.

TCP assumes the datagram service it is layered above is unreliable. A checksum over all data helps TCP implement reliability. Using a window-based flow control mechanism that makes use of positive acknowledgements, sequence numbers, and a retransmission strategy, TCP can usually recover when datagrams are damaged, delayed, duplicated or delivered out of order by the underlying communication medium.

If the local TCP receives no acknowledgements from its peer for a period of time, as would be the case if the remote machine crashed, the connection is closed and an error is returned to the user. If the remote machine reboots or otherwise loses state information about a TCP connection, the connection is aborted and an error is returned to the user.

ERRORS

A socket operation may fail if:

EISCONN	A **connect** operation was attempted on a socket on which a **connect** operation had already been performed.
ETIMEDOUT	A connection was dropped due to excessive retransmissions.
ECONNRESET	The remote peer forced the connection to be closed (usually because the remote machine has lost state information about the connection due to a crash).
ECONNREFUSED	The remote peer actively refused connection establishment (usually because no process is listening to the port).
EADDRINUSE	A **bind** operation was attempted on a socket with a network address/port pair that has already been bound to another socket.
EADDRNOTAVAIL	A **bind** operation was attempted on a socket with a network address for which no network interface exists.
EACCES	A **bind** operation was attempted with a "reserved" port number and the effective user ID of the process was not super-user.
ENOBUFS	The system ran out of memory for internal data structures.

SEE ALSO

accept(2), **bind**(2), **connect**(2), **getsockopt**(2), **listen**(2), **read**(2V), **send**(2), **write**(2V), **getprotoent**(3N), **inet**(4F), **ip**(4P)

Postel, Jon, *Transmission Control Protocol - DARPA Internet Program Protocol Specification*, RFC 793, Network Information Center, SRI International, Menlo Park, Calif., September 1981.

BUGS

SIOCSHIWAT and **SIOCGHIWAT** ioctl's to set and get the high water mark for the socket queue, and so that it can be changed from 2048 bytes to be larger or smaller, have been defined (in **<sys/ioctl.h>**) but not implemented.

NAME

tcptli – TLI-Conforming TCP Stream-Head

CONFIG

pseudo-device clone

pseudo-device tcptli32

SYNOPSIS

```
#include <fcntl.h>
#include <nettli/tiuser.h>

tfd = t_open("/dev/tcp", O_RDWR, tinfo);
struct t_info *tinfo;
```

DESCRIPTION

TCPTLI provides access to TCP service via the Transport Library Interface (TLI). Prior to this release, TCP access was only possible via the socket programming interface. Programmers have the choice of using either the socket or TLI programming interface for their application.

TCPTLI is implemented in STREAMS conforming to the Transport Provider Interface (TPI) specification as a TCP Transport Provider to a TLI application. It utilizes the existing underlying socket and TCP support in the SunOS kernel to communicate over the network. It is also a clone driver, see **clone(4)** for more characteristics pertaining to a clone STREAMS driver.

The notion of an address is the same as the socket address (struct sockaddr_in) defined in <netinet/in.h>. TCPTLI maintains transport state information for each outstanding connection and the current state of the provider may be retrieved via the **t_getstate(3N)** call. See **t_getstate(3N)** for a list of possible states.

A server usually starts up with the **t_open(3N)** call followed by **t_bind(3N)** to bind an address that it listens for incoming connection. It may call **t_listen(3N)** to retrieve an indication of a connect request from another transport user, and then calls **t_accept(3N)** if it is willing to provide its service. TLI allows a server to accept connection on the same file descriptor it is listening on, or a different file descriptor (as in the sense of socket's **accept(2)**).

A client usually calls **t_open(3N)** and followed by a call to **t_bind(3N)**. Then it calls **t_connect(3N)** to the address of a server advertized for providing service. Once the connection is established, it may use **t_rcv(3N)** and **t_snd(3N)** to receive and send data. The routine **t_close(3N)** is used to terminate the connection.

TLI ERRORS

An TLI operation may fail if one of the following error conditions is encountered. They are returned by the TLI user level library.

TBADADDR	Incorrect/invalid address format supplied by the user.
TBADOPT	Incorrect option.
TACCESS	No permission.
TBADF	Illegal transport file descriptor.
TNOADDR	Could not allocate address
TOUTSTATE	The transport is in an incorrect state.
TBADSEQ	Incorrect sequence number.
TSYSERR	A system error, i.e. below the transport level (see list below) is encountered.
TLOOK	An event requires attention.
TBADDATA	Illegal amount of data
TBUFOVFLW	Buffer not large enough.

TFLOW	Flow control problem.
TNODATA	No data.
TNODIS	No discon_ind is found on the queue.
TNOUDERR	Unit data not found.
TBADFLAG	Bad flags.
TNOREL	No orderly release request found on queue.
TNOTSUPPORT	Protocol/primitive is not supported.
TSTATECHNG	State is in the process of changing.

SYSTEM ERRORS

The following errors are returned by TCPTLI. However they may be translated to the above TLI errors by the user level library (**libnsl**).

ENXIO	Invalid device or address, out of range.
EBUSY	Request device is busy or not ready.
ENOMEM	Not enough memory for transmitting data, non fatal.
EPROTO	The operation encountered an underlying protocol. error (TCP).
EWOULDBLOCK	The operation would block as normally the file descriptors are set with non-blocking flag.
EACCES	Permission denied.
ENOBUFS	The system ran out of memory for internal (network) data structures.

SEE ALSO

accept(2), **t_open**(3N), **t_close**(3N), **t_accept**(3N), **t_getstate**(3N), **t_bind**(3N), **t_connect**(3N), **t_rcv**(3N), **t_snd**(3N), **t_alloc**(3N), **t_unbind**(3N), **t_getinfo**(3N)

BUGS

Only TCP (i.e. connection oriented) protocol is supported, no UDP. The maximum network connection is 32 by default. A new kernel has to be configured if an increase of such limit is desired: by changing the entry **pseudo-device tcptli32** in the kernel config file to **tcptli64.**

NAME

termio – general terminal interface

SYNOPSIS

#include <sys/termios.h>

DESCRIPTION

Asynchronous communications ports, pseudo-terminals, and the special interface accessed by **/dev/tty** all use the same general interface, no matter what hardware (if any) is involved. The remainder of this section discusses the common features of this interface.

Opening a Terminal Device File

When a terminal file is opened, the process normally waits until a connection is established. In practice, users' programs seldom open these files; they are opened by **getty**(8) and become a user's standard input, output, and error files. The state of the software carrier flag will effect the ability to open a line.

Sessions

Processes are now grouped by session, then process group, then process id. Each session is associated with one "login" session (windows count as logins). A process creates a session by calling **setsid**(2V), which will put the process in a new session as its only member and as the session leader of that session.

Process Groups

A terminal may have a distinguished process group associated with it. This distinguished process group plays a special role in handling signal-generating input characters, as discussed below in the **Special Characters** section below. The terminal's process group can can be set only to process groups that are members of the terminal's session.

A command interpreter, such as **csh**(1), that supports "job control" can allocate the terminal to different *jobs*, or process groups, by placing related processes in a single process group and associating this process group with the terminal. A terminal's associated process group may be set or examined by a process with sufficient privileges. The terminal interface aids in this allocation by restricting access to the terminal by processes that are not in the current process group; see **Job Access Control** below.

Orphaned Process Groups

An orphaned process group is a process group that has no parent, in a different process group, and in the same session. In other words, there is no process that can handle job control signals for the process group.

The Controlling Terminal

A terminal may belong to a process as its *controlling terminal*. If a process that is a session leader, and that does not have a controlling terminal, opens a terminal file not already associated with a session, the terminal associated with that terminal file becomes the controlling terminal for that process, and the terminal's distinguished process group is set to the process group of that process. (Currently, this also happens if a process that does not have a controlling terminal and is not a member of a process group opens a terminal. In this case, if the terminal is not associated with a session, a new session is created with a process group ID equal to the process ID of the process in question, and the terminal is assigned to that session. The process is made a member of the terminal's process group.)

If a process does not wish to acquire the terminal as a controlling terminal (as is the case with many daemons that open **/dev/console**), the process should or O_NOCTTY into the second argument to **open**(2V).

The controlling terminal is inherited by a child process during a **fork**(2V). A process relinquishes its control terminal when it changes its process group using **setsid**(2V), when it trys to change back to process group 0 via a **setpgrp**(2V) with arguments (**mypid, 0**), or when it issues a TIOCNOTTY **ioctl**(2) call on a file descriptor created by opening the file **/dev/tty**. Both of the last two cases cause a **setsid**(2V) to be called on the process' behalf. This is an attempt to allow old binaries (that couldn't have known about **setsid**(2V)) to still acquire controlling terminals. It doesn't always work, see **setsid**(8V) for a workaround for those cases.

When a session leader that has a controlling terminal terminates, the distinguished process group of the controlling terminal is set to zero (indicating no distinguished process group). This allows the terminal to be acquired as a controlling terminal by a new session leader.

Closing a Terminal Device File

When a terminal device file is closed, the process closing the file waits until all output is drained; all pending input is then flushed, and finally a disconnect is performed. If **HUPCL** is set, the existing connection is severed (by hanging up the phone line, if appropriate).

Job Access Control

If a process is in the (non-zero) distinguished process group of its controlling terminal (if this is true, the process is said to be a *foreground process*), then **read**(2V) operations are allowed as described below in **Input Processing and Reading Characters**. If a process is not in the (non-zero) distinguished process group of its controlling terminal (if this is true, the process is said to be a *background process*), then any attempts to read from that terminal will typically send that process' process group a **SIGTTIN** signal. If the process is ignoring **SIGTTIN**, has **SIGTTIN** blocked, is a member of an orphaned process group, or is in the middle of process creation using **vfork**(2), the read will return −1 and set **errno** to **EIO**, and the **SIGTTIN** signal will not be sent. The **SIGTTIN** signal will normally stop the members of that process group.

When the **TOSTOP** bit is set in the **c_lflag** field, attempts by a background process to write to its controlling terminal will typically send that process' process group a **SIGTTOU** signal. If the process is ignoring **SIGTTOU**, has **SIGTTOU** blocked, or is in the middle of process creation using **vfork**(), the process will be allowed to write to the terminal and the **SIGTTOU** signal will not be sent. If the process is orphaned, the write will return −1 and set **errno** to **EIO**, and the **SIGTTOU** signal will not be sent. **SIGTTOU** signal will normally stop the members of that process group. Certain **ioctl**() calls that set terminal parameters are treated in this same fashion, except that **TOSTOP** is not checked; the effect is identical to that of terminal writes when **TOSTOP** is set. See **IOCTLS**.

Input Processing and Reading Characters

A terminal associated with one of these files ordinarily operates in full-duplex mode. Characters may be typed at any time, even while output is occurring, and are only lost when the system's character input buffers become completely full, which is rare, or when the user has accumulated the maximum allowed number of input characters that have not yet been read by some program. This limit is available is {MAX_CANON} characters (see **pathconf**(2V)). If the **IMAXBEL** mode has not been selected, all the saved characters are thrown away without notice when the input limit is reached; if the **IMAXBEL** mode has been selected, the driver refuses to accept any further input, and echoes a bell (ASCII BEL).

Two general kinds of input processing are available, determined by whether the terminal device file is in canonical mode or non-canonical mode (see **ICANON** in the **Local Modes** section).

The style of input processing can also be very different when the terminal is put in non-blocking I/O mode; see **read**(2V). In this case, reads from the terminal will never block.

It is possible to simulate terminal input using the **TIOCSTI ioctl**() call, which takes, as its third argument, the address of a character. The system pretends that this character was typed on the argument terminal, which must be the process' controlling terminal unless the process' effective user ID is super-user.

Canonical Mode Input Processing

In canonical mode input processing, terminal input is processed in units of lines. A line is delimited by a NEWLINE (ASCII LF) character, an EOF (by default, an ASCII EOT) character, or one of two user-specified end-of-line characters, **EOL** and **EOL2**. This means that a **read**() will not complete 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 may be requested in a read, even one, without losing information.

Erase and kill processing occurs during input. The **ERASE** character (by default, the character DEL) erases the last character typed in the current input line. The **WERASE** character (by default, the character CTRL-W) erases the last "word" typed in the current input line (but not any preceding SPACE or TAB characters). A "word" is defined as a sequence of non-blank characters, with TAB characters counted as blanks.

Neither **ERASE** nor **WERASE** will erase beyond the beginning of the line. The **KILL** character (by default, the character CTRL-U) kills (deletes) the entire current input line, and optionally outputs a NEWLINE character. All these characters operate on a key-stroke basis, independently of any backspacing or tabbing that may have been done.

The **REPRINT** character (the character CTRL-R) prints a NEWLINE followed by all characters that have not been read. Reprinting also occurs automatically if characters that would normally be erased from the screen are fouled by program output. The characters are reprinted as if they were being echoed; as a consequence, if **ECHO** is not set, they are not printed.

The **ERASE** and **KILL** characters may be entered literally by preceding them with the escape character (\). In this case the escape character is not read. The **ERASE** and **KILL** characters may be changed.

Non-Canonical Mode Input Processing

In non-canonical mode input processing, input characters are not assembled into lines, and erase and kill processing does not occur. The MIN and TIME values are used to determine how to process the characters received.

MIN represents the minimum number of characters that should be received when the read is satisfied (when the characters are returned to the user). TIME is a timer of 0.10 second granularity that is used to timeout bursty and short term data transmissions. The four possible values for MIN and TIME and their interactions are described below.

Case A: MIN > 0, TIME > 0

In this case TIME serves as an intercharacter timer and is activated after the first character is received. Since it is an intercharacter timer, it is reset after a character is received. The interaction between MIN and TIME is as follows: as soon as one character is received, the intercharacter timer is started. If MIN characters are received before the intercharacter 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: if MIN 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 will sleep until the MIN and TIME mechanisms are activated by the receipt of the first character.

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 (the pending read will sleep until MIN characters are received). A program that uses this case to read record-based terminal I/O may block indefinitely in the read operation.

Case C: MIN = 0, TIME > 0

In this case, since MIN = 0, TIME no longer represents an intercharacter timer. It now serves as a read timer that is activated as soon as a **read()** is done. A read is satisfied as soon as a single character is received or the read timer expires. Note: in this case if the timer expires, no character will be returned. If the timer does not expire, the only way the read can be satisfied is if a character is received. In this case the read will not block indefinitely waiting for a character – if no character is received within TIME*.10 seconds after the read is initiated, the read will return with zero characters.

Case D: MIN = 0, TIME = 0

In this case return is immediate. The minimum of either the number of characters requested or the number of characters currently available will be returned without waiting for more characters to be input.

Comparison of the Different Cases of MIN, TIME Interaction

Some points to note about MIN and TIME:

- In the following explanations one may notice that 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.

- Also note that in case A (**MIN** > 0, **TIME** > 0), **TIME** represents an intercharacter timer while in case C (**TIME** = 0, **TIME** > 0) **TIME** represents a read timer.

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 (for example, file transfer programs) where a program would like to process at least **MIN** characters at a time. In case A, the intercharacter 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. That is, if a program does a read of 20 bytes, **MIN** is 10, and 25 characters are present, 20 characters will be returned to the user.

Writing Characters

When one or more characters are written, they are transmitted to the terminal as soon as previously-written characters have finished typing. Input characters are echoed as they are typed if echoing has been enabled. If a process produces characters more rapidly than they can be typed, it will be suspended when its output queue exceeds some limit. When the queue has drained down to some threshold, the program is resumed.

Special Characters

Certain characters have special functions on input and/or output. These functions and their default character values are summarized as follows:

INTR (CTRL-C or ASCII ETX) generates a **SIGINT** signal, which is sent to all processes in the distinguished process group associated with the terminal. Normally, each such process is forced to terminate, but arrangements may be made either to ignore the signal or to receive a trap to an agreed-upon location; see **sigvec**(2).

QUIT (CTRL-\ or ASCII FS) generates a **SIGQUIT** signal, which is sent to all processes in the distinguished process group associated with the terminal. Its treatment is identical to the interrupt signal except that, unless a receiving process has made other arrangements, it will not only be terminated but a core image file (called **core**) will be created in the current working directory.

ERASE (Rubout or ASCII DEL) erases the preceding character. It will not erase beyond the start of a line, as delimited by a **NL**, **EOF**, **EOL**, or **EOL2** character.

WERASE (CTRL-W or ASCII ETB) erases the preceding "word". It will not erase beyond the start of a line, as delimited by a **NL**, **EOF**, **EOL**, or **EOL2** character.

KILL (CTRL-U or ASCII NAK) deletes the entire line, as delimited by a **NL**, **EOF**, **EOL**, or **EOL2** character.

REPRINT (CTRL-R or ASCII DC2) reprints all characters that have not been read, preceded by a NEWLINE.

EOF (CTRL-D or ASCII EOT) may 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 NEWLINE, and the **EOF** is discarded. Thus, if there are no characters waiting, which is to say the **EOF** occurred at the beginning of a line, zero characters will be passed back, which is the standard end-of-file indication.

NL (ASCII LF) is the normal line delimiter. It can not be changed; it can, however, be escaped by the **LNEXT** character.

EOL
EOL2 (ASCII NUL) are additional line delimiters, like **NL**. They are not normally used.

SUSP (CTRL-Z or ASCII EM) is used by the job control facility to change the current job to return to the controlling job. It generates a **SIGTSTP** signal, which stops all processes in the terminal's process group.

STOP (CTRL-S or ASCII DC3) can be used to temporarily suspend output. It is useful with CRT terminals to prevent output from disappearing before it can be read. While output is suspended, **STOP** characters are ignored and not read.

START (CTRL-Q or ASCII DC1) is used to resume output that has been suspended by a **STOP** character. While output is not suspended, **START** characters are ignored and not read.

DISCARD (CTRL-O or ASCII SI) causes subsequent output to be discarded until another **DISCARD** character is typed, more input arrives, or the condition is cleared by a program.

LNEXT (CTRL-V or ASCII SYN) causes the special meaning of the next character to be ignored; this works for all the special characters mentioned above. This allows characters to be input that would otherwise get interpreted by the system (for example, **KILL**, **QUIT**.)

The character values for **INTR**, **QUIT**, **ERASE**, **WERASE**, **KILL**, **REPRINT**, **EOF**, **EOL**, **EOL2**, **SUSP**, **STOP**, **START**, **DISCARD**, and **LNEXT** may be changed to suit individual tastes. If the value of a special control character is 0, the function of that special control character will be disabled. The **ERASE**, **KILL**, and **EOF** characters may be escaped by a preceding \ character, in which case no special function is done. Any of the special characters may be preceded by the **LNEXT** character, in which case no special function is done.

If **IEXTEN** is added to the local modes (this is the default), then all of the special characters are in effect. If **IEXTEN** is cleared from the local modes, then only the following POSIX.1 compatible specials are seen as specials: **INTR**, **QUIT**, **ERASE**, **KILL**, **EOF**, **NL**, **EOL**, **SUSP**, **STOP**, **START**, and **CR**.

Software Carrier Mode

The software carrier mode can be enabled or disabled using the **TIOCSSOFTCAR** **ioctl()**. If the software carrier flag for a line is off, the line pays attention to the hardware carrier detect (DCD) signal. The **tty** device associated with the line can not be opened until DCD is asserted. If the software carrier flag is on, the line behaves as if DCD is always asserted.

The software carrier flag is usually turned on for locally connected terminals or other devices, and is off for lines with modems.

To be able to issue the **TIOCGSOFTCAR** and **TIOCSSOFTCAR** **ioctl()** calls, the **tty** line should be opened with **O_NDELAY** so that the **open**(2V) will not wait for the carrier.

Modem Disconnect

If a modem disconnect is detected, and the **CLOCAL** flag is not set in the **c_cflag** field, a **SIGHUP** signal is sent to all processes in the distinguished process group associated with this terminal. Unless other arrangements have been made, this signal terminates the processes. If **SIGHUP** is ignored or caught, any subsequent **read()** returns with an end-of-file indication until the terminal is closed. Thus, programs that read a terminal and test for end-of-file can terminate appropriately after a disconnect. Any subsequent **write()** will return −1 and set **errno** to **EIO** until the terminal is closed.

A **SIGHUP** signal is sent to the **tty** if the software carrier flag is off and the hardware carrier detect drops.

Terminal Parameters

The parameters that control the behavior of devices and modules providing the **termios** interface are specified by the **termios** structure, defined by **<sys/termios.h>**. Several **ioctl()** system calls that fetch or change these parameters use this structure:

```
#define  NCCS     17
struct   termios {
         unsigned long  c_iflag;    /* input modes */
         unsigned long  c_oflag;    /* output modes */
         unsigned long  c_cflag;    /* control modes */
```

```
        unsigned  long   c_lflag;      /* local modes */
        unsigned  char   c_line;       /* line discipline */
        unsigned  char   c_cc[NCCS]; /* control chars */
};
```

The special control characters are defined by the array **c_cc**. The relative positions and initial values for each function are as follows:

0	VINTR	ETX
1	VQUIT	FS
2	VERASE	DEL
3	VKILL	NAK
4	VEOF	EOT
5	VEOL	NUL
6	VEOL2	NUL
7	VSWTCH	NUL
8	VSTART	DC1
9	VSTOP	DC3
10	VSUSP	EM
12	VREPRINT	DC2
13	VDISCARD	SI
14	VWERASE	ETB
15	VLNEXT	SYN

The **MIN** value is stored in the **VMIN** element of the **c_cc** array, and the **TIME** value is stored in the **VTIME** element of the **c_cc** array. The **VMIN** element is the same element as the **VEOF** element, and the **VTIME** element is the same element as the **VEOL** element.

Input Modes

The **c_iflag** field describes the basic terminal input control:

IGNBRK	0000001	Ignore break condition.
BRKINT	0000002	Signal interrupt on break.
IGNPAR	0000004	Ignore characters with parity errors.
PARMRK	0000010	Mark parity errors.
INPCK	0000020	Enable input parity check.
ISTRIP	0000040	Strip character.
INLCR	0000100	Map NL to CR on input.
IGNCR	0000200	Ignore CR.
ICRNL	0000400	Map CR to NL on input.
IUCLC	0001000	Map upper-case to lower-case on input.
IXON	0002000	Enable start/stop output control.
IXANY	0004000	Enable any character to restart output.
IXOFF	0010000	Enable start/stop input control.
IMAXBEL	0020000	Echo BEL on input line too long.

If **IGNBRK** is set, a break condition (a character framing error with data all zeros) detected on input is ignored, that is, not put on the input queue and therefore not read by any process. Otherwise, if **BRKINT** is set, a break condition will generate a **SIGINT** and flush both the input and output queues. If neither **IGNBRK** nor **BRKINT** is set, a break condition is read as a single ASCII NUL character ('\0').

If **IGNPAR** is set, characters with framing or parity errors (other than break) are ignored. Otherwise, if **PARMRK** is set, a character with a framing or parity error that is not ignored 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 not set, a valid character of '\377' is read as '\377', '\377'. If neither **IGNPAR** nor **PARMRK** is set, a framing or parity error (other than break) is read as a single ASCII NUL character ('\0').

If **INPCK** is set, input parity checking is enabled. If **INPCK** is not set, input parity checking is disabled. This allows output parity generation without input parity errors.

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). Otherwise if **ICRNL** is set, a received CR character is translated into a NL character.

If **IUCLC** is set, a received upper-case alphabetic character is translated into the corresponding lower-case character.

If **IXON** is set, start/stop output control is enabled. A received **STOP** character will suspend output and a received **START** character will restart output. The **STOP** and **START** characters will not be read, but will merely perform flow control functions. If **IXANY** is set, any input character will restart output that has been suspended.

If **IXOFF** is set, the system will transmit a **STOP** character when the input queue is nearly full, and a **START** character when enough input has been read that the input queue is nearly empty again.

If **IMAXBEL** is set, the ASCII BEL character is echoed if the input stream overflows. Further input will not be stored, but any input already present in the input stream will not be disturbed. If **IMAXBEL** is not set, no BEL character is echoed, and all input present in the input queue is discarded if the input stream overflows.

The initial input control value is **BRKINT, ICRNL, IXON, ISTRIP.**

Output modes

The **c_oflag** field specifies the system treatment of output:

OPOST	0000001	Postprocess output.
OLCUC	0000002	Map lower case to upper on output.
ONLCR	0000004	Map NL to CR-NL on output.
OCRNL	0000010	Map CR to NL on output.
ONOCR	0000020	No CR output at column 0.
ONLRET	0000040	NL performs CR function.
OFILL	0000100	Use fill characters for delay.
OFDEL	0000200	Fill is DEL, else NUL.
NLDLY	0000400	Select new-line delays:
NL0	0	
NL1	0000400	
CRDLY	0003000	Select carriage-return delays:
CR0	0	
CR1	0001000	
CR2	0002000	
CR3	0003000	
TABDLY	0014000	Select horizontal-tab delays:
TAB0	0	or tab expansion:
TAB1	0004000	
TAB2	0010000	
XTABS	0014000	Expand tabs to spaces.
BSDLY	0020000	Select backspace delays:
BS0	0	
BS1	0020000	
VTDLY	0040000	Select vertical-tab delays:
VT0	0	
VT1	0040000	

FFDLY	0100000	Select form-feed delays:
FF0	0	
FF1	0100000	

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 lower-case alphabetic character is transmitted as the corresponding upper-case 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. Otherwise the NL character is assumed to do just the line-feed function; the column pointer will remain unchanged. The column pointer is also 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. In all cases a value of 0 indicates no delay. If **OFILL** is set, fill characters will be 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, the RETURN delays are used instead of the NEWLINE delays. If **OFILL** is set, two fill characters will be transmitted.

Carriage-return delay type 1 is dependent on the current column position, type 2 is about 0.10 seconds, and type 3 is about 0.15 seconds. If **OFILL** is set, delay type 1 transmits two fill characters, and type 2, four fill characters.

Horizontal-tab delay type 1 is dependent on the current column position. Type 2 is about 0.10 seconds. Type 3, specified by **TAB3** or **XTABS**, specifies that TAB characters are to be expanded into SPACE characters. If **OFILL** is set, two fill characters will be transmitted for any delay.

Backspace delay lasts about 0.05 seconds. If **OFILL** is set, one fill character will be transmitted.

The actual delays depend on line speed and system load.

The initial output control value is **OPOST, ONLCR, XTABS**.

The **c_cflag** field describes the hardware control of the terminal:

CBAUD	0000017	Baud rate:
B0	0	Hang up
B50	0000001	50 baud
B75	0000002	75 baud
B110	0000003	110 baud
B134	0000004	134.5 baud
B150	0000005	150 baud
B200	0000006	200 baud
B300	0000007	300 baud
B600	0000010	600 baud
B1200	0000011	1200 baud
B1800	0000012	1800 baud
B2400	0000013	2400 baud
B4800	0000014	4800 baud
B9600	0000015	9600 baud
B19200	0000016	19200 baud
B38400	0000017	38400 baud

CSIZE	0000060	Character size:
CS5	0	5 bits
CS6	0000020	6 bits
CS7	0000040	7 bits
CS8	0000060	8 bits
CSTOPB	0000100	Send two stop bits, else one.
CREAD	0000200	Enable receiver.
PARENB	0000400	Parity enable.
PARODD	0001000	Odd parity, else even.
HUPCL	0002000	Hang up on last close.
CLOCAL	0004000	Local line, else dial-up.
CIBAUD	03600000	Input baud rate, if different from output rate.
CRTSCTS	020000000000	Enable RTS/CTS flow control.

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 will cease to be asserted. Normally, this will disconnect the line. If the **CIBAUD** bits are not zero, they specify the input baud rate, with the **CBAUD** bits specifying the output baud rate; otherwise, the output and input baud rates are both specified by the **CBAUD** bits. The values for the **CIBAUD** bits are the same as the values for the **CBAUD** bits, shifted left **IBSHIFT** bits. For any particular hardware, impossible speed changes are ignored.

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, two stop bits are required.

If **PARENB** is set, parity generation and detection is enabled and a parity bit is added to each character. If parity is enabled, the **PARODD** flag specifies odd parity if set, otherwise even parity is used.

If **CREAD** is set, the receiver is enabled. Otherwise no characters will be received.

If **HUPCL** is set, the modem control lines for the port will be disconnected when the last process with the line open closes it or terminates.

If **CLOCAL** is set, a connection does not depend on the state of the modem status lines. Otherwise modem control is assumed.

If **CRTSCTS** is set, and the terminal has modem control lines associated with it, the Request To Send (RTS) modem control line will be raised, and output will occur only if the Clear To Send (CTS) modem status line is raised. If the CTS modem status line is lowered, output is suspended until CTS is raised. Some hardware may not support this function, and other hardware may not permit it to be disabled; in either of these cases, the state of the **CRTSCTS** flag is ignored.

The initial hardware control value after open is **B9600**, **CS7**, **CREAD**, **PARENB**.

Local Modes

The **c_lflag** field of the argument structure is used by the line discipline to control terminal functions. The basic line discipline provides the following:

ISIG	0000001	Enable signals.
ICANON	0000002	Canonical input (erase and kill processing).
XCASE	0000004	Canonical upper/lower presentation.
ECHO	0000010	Enable echo.
ECHOE	0000020	Echo erase character as BS-SP-BS.
ECHOK	0000040	Echo NL after kill character.
ECHONL	0000100	Echo NL.
NOFLSH	0000200	Disable flush after interrupt or quit.
TOSTOP	0000400	Send **SIGTTOU** for background output.
ECHOCTL	0001000	Echo control characters as ^*char*, delete as ^?.
ECHOPRT	0002000	Echo erase character as character erased.
ECHOKE	0004000	BS-SP-BS erase entire line on line kill.

FLUSHO	0020000	Output is being flushed.
PENDIN	0040000	Retype pending input at next read or input character.
IEXTEN	0100000	Recognize all specials (if clear, POSIX only).

If **ISIG** is set, each input character is checked against the special control characters **INTR**, **QUIT**, and **SUSP**. If an input character matches one of these control characters, the function associated with that character is performed. If **ISIG** is not set, no checking is done. Thus these special input functions are possible only if **ISIG** is set.

If **ICANON** is set, canonical processing is enabled. This is affected by the **IEXTEN** bit (see **Special Characters** above). This enables the erase, word erase, kill, and reprint edit functions, and the assembly of input characters into lines delimited by **NL**, **EOF**, **EOL**, and **EOL2**. If **ICANON** is not set, read requests are satisfied directly from the input queue. A read will not be satisfied until at least **MIN** characters have been received or the timeout value **TIME** has expired between characters. This allows fast bursts of input to be read efficiently while still allowing single character input. The time value represents tenths of seconds. See the *Non-canonical Mode Input Processing* section for more details.

If **XCASE** is set, and if **ICANON** is set, an upper-case 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:

for:	*use*:
`	\'
|	\!
~	\^
{	\(
}	\)
\	\\

For example, **A** is input as **\a**, **\n** as **\\n**, and **\N** as **\\\n**.

If **ECHO** is set, characters are echoed as received. If **ECHO** is not set, input characters are not echoed.

If **ECHOCTL** is not set, all control characters (characters with codes between 0 and 37 octal) are echoed as themselves. If **ECHOCTL** is set, all control characters other than ASCII TAB, ASCII NL, the **START** character, and the **STOP** character, are echoed as ^*X*, where *X* is the character given by adding 100 octal to the control character's code (so that the character with octal code 1 is echoed as '^A'), and the ASCII DEL character, with code 177 octal, is echoed as '^?'.

When **ICANON** is set, the following echo functions are possible:

- If **ECHO** and **ECHOE** are set, and **ECHOPRT** is not set, the **ERASE** and **WERASE** characters are echoed as one or more ASCII BS SP BS, which will clear the last character(s) from a CRT screen.
- If **ECHO** and **ECHOPRT** are set, the first **ERASE** and **WERASE** character in a sequence echoes as a backslash (\) followed by the characters being erased. Subsequent **ERASE** and **WERASE** characters echo the characters being erased, in reverse order. The next non-erase character types a slash (/) before it is echoed.
- If **ECHOKE** is set, the kill character is echoed by erasing each character on the line from the screen (using the mechanism selected by **ECHOE** and **ECHOPRT**).
- If **ECHOK** is set, and **ECHOKE** is not set, the NL character will be echoed after the kill character to emphasize that the line will be deleted. Note: an escape character (\) or an **LNEXT** character preceding the erase or kill character removes any special function.
- If **ECHONL** is set, the NL character will be echoed even if **ECHO** is not set. This is useful for terminals set to local echo (so-called half duplex).

- If ECHOCTL is not set, the EOF character is not echoed, unless it is escaped. Because EOT is the default EOF character, this prevents terminals that respond to EOT from hanging up. If ECHOCTL is set, the EOF character is echoed; if it is not escaped, after it is echoed, one backspace character is output if it is echoed as itself, and two backspace characters are echoed if it is echoed as ^X.

If NOFLSH is set, the normal flush of the input and output queues associated with the INTR, QUIT, and SUSP characters will not be done.

If TOSTOP is set, the signal SIGTTOU is sent to a process that tries to write to its controlling terminal if it is not in the distinguished process group for that terminal. This signal normally stops the process. Otherwise, the output generated by that process is output to the current output stream. Processes that are blocking or ignoring SIGTTOU signals are excepted and allowed to produce output.

If FLUSHO is set, data written to the terminal will be discarded. This bit is set when the FLUSH character is typed. A program can cancel the effect of typing the FLUSH character by clearing FLUSHO.

If PENDIN is set, any input that has not yet been read will be reprinted when the next character arrives as input.

The initial line-discipline control value is ISIG, ICANON, ECHO.

Minimum and Timeout

The MIN and TIME values are described above under **Non-canonical Mode Input Processing**. The initial value of MIN is 1, and the initial value of TIME is 0.

Termio Structure

The System V **termio** structure is used by other **ioctl()** calls; it is defined by **<sys/termio.h>** as:

```
#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 */
};
```

The special control characters are defined by the array **c_cc**. The relative positions for each function are as follows:

0	**VINTR**
1	**VQUIT**
2	**VERASE**
3	**VKILL**
4	**VEOF**
5	**VEOL**
6	**VEOL2**
7	reserved

The calls that use the **termio** structure only affect the flags and control characters that can be stored in the **termio** structure; all other flags and control characters are unaffected.

Terminal Size

The number of lines and columns on the terminal's display (or page, in the case of printing terminals) is specified in the **winsize** structure, defined by **<sys/termios.h>**. Several **ioctl()** system calls that fetch or change these parameters use this structure:

```
struct winsize {
        unsigned short  ws_row;     /* rows, in characters */
        unsigned short  ws_col;     /* columns, in characters */
```

```
        unsigned short   ws_xpixel;   /* horizontal size, pixels - not used */
        unsigned short   ws_ypixel;   /* vertical size, pixels - not used */
};
```

Modem Lines

On special files representing serial ports, the modem control lines supported by the hardware can be read and the modem status lines supported by the hardware can be changed. The following modem control and status lines may be supported by a device; they are defined by **<sys/termios.h>**:

TIOCM_LE	0001	line enable
TIOCM_DTR	0002	data terminal ready
TIOCM_RTS	0004	request to send
TIOCM_ST	0010	secondary transmit
TIOCM_SR	0020	secondary receive
TIOCM_CTS	0040	clear to send
TIOCM_CAR	0100	carrier detect
TIOCM_RNG	0200	ring
TIOCM_DSR	0400	data set ready

TIOCM_CD is a synonym for **TIOCM_CAR**, and **TIOCM_RI** is a synonym for **TIOCM_RNG**.

Not all of these will necessarily be supported by any particular device; check the manual page for the device in question.

IOCTLS

The **ioctl()** calls supported by devices and STREAMS modules providing the **termios** interface are listed below. Some calls may not be supported by all devices or modules.

Unless otherwise noted for a specific **ioctl()** call, these functions are restricted from use by background processes. Attempts to perform these calls will cause the process group of the process performing the call to be sent a **SIGTTOU** signal. If the process is ignoring **SIGTTOU**, has **SIGTTOU** blocked, or is in the middle of process creation using **vfork()**, the process will be allowed to perform the call and the **SIGTTOU** signal will not be sent.

TCGETS The argument is a pointer to a **termios** structure. The current terminal parameters are fetched and stored into that structure. This call is allowed from a background process; however, the information may subsequently be changed by a foreground process.

TCSETS The argument is a pointer to a **termios** structure. The current terminal parameters are set from the values stored in that structure. The change is immediate.

TCSETSW The argument is a pointer to a **termios** 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 will affect output.

TCSETSF The argument is a pointer to a **termios** 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 and then the change occurs.

TCGETA The argument is a pointer to a **termio** structure. The current terminal parameters are fetched, and those parameters that can be stored in a **termio** structure are stored into that structure. This call is allowed from a background process; however, the information may subsequently be changed by a foreground process.

TCSETA The argument is a pointer to a **termio** structure. Those terminal parameters that can be stored in a **termio** structure are set from the values stored in that structure. The change is immediate.

TCSETAW The argument is a pointer to a **termio** structure. Those terminal parameters that can be stored in a **termio** structure 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 will affect output.

TCSETAF The argument is a pointer to a **termio** structure. Those terminal parameters that can be stored in a **termio** structure 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 and then the change occurs.

TCSBRK The argument is an **int** value. Wait for the output to drain. If the argument is 0, then send a break (zero-valued bits for 0.25 seconds). This define is available by **#include <sys/termio.h>**

TCXONC Start/stop control. The argument is an **int** value. If the argument is **TCOOFF** (0), suspend output; if **TCOON** (1), restart suspended output; if **TCIOFF** (2), suspend input; if **TCION** (3), restart suspended input.

TCFLSH The argument is an **int** value. If the argument is **TCIFLUSH** (0), flush the input queue; if **TCOFLUSH** (1), flush the output queue; if **TCIOFLUSH** (2), flush both the input and output queues.

TIOCEXCL The argument is ignored. Exclusive-use mode is turned on; no further opens are permitted until the file has been closed, or a **TIOCNXCL** is issued. The default on open of a terminal file is that exclusive use mode is off. This **ioctl()** is only available by **#include <sys/ttold.h>** .

TIOCNXCL The argument is ignored. Exclusive-use mode is turned off. This **ioctl()** is only available by **#include <sys/ttold.h>**.

TIOCSCTTY The argument is an **int**. The system will attempt to assign the terminal as the caller's controlling terminal (see **The Controlling Terminal** above). If the caller is not the super-user and/or the argument is not 1, all of the normal permission checks apply. If the caller is the super-user and the argument is 1 the terminal will be assigned as the controlling terminal even if the terminal was currently in use as a controlling terminal by another session. **getty**(8) uses this method to acquire controlling terminals for **login**(1) because there exists a possibility that a daemon process may obtain the console before **getty**(8).

TIOCGPGRP The argument is a pointer to an **int**. Set the value of that **int** to the process group ID of the distinguished process group associated with the terminal. This call is allowed from a background process; however, the information may subsequently be changed by a foreground process. This **ioctl()** exists only for backward compatibility, use **tcgetpgrp**(3V).

TIOCSPGRP The argument is a pointer to an **int**. Associate the process group whose process group ID is specified by the value of that **int** with the terminal. The new process group value must be in the range of valid process group ID values, or it must be zero ("no process group"). Otherwise, the error **EINVAL** is returned. If any processes exist with a process ID or process group ID that is the same as the new process group value, then those processes must have the same real or saved user ID as the real or effective user ID of the calling process or be descendants of the calling process, or the effective user ID of the current process must be super-user. Otherwise, the error **EPERM** is returned. This **ioctl()** exists only for backward compatibility, use **tcsetpgrp()**, see **tcgetpgrp**(3V).

TIOCOUTQ The argument is a pointer to an **int**. Set the value of that **int** to the number of characters in the output stream that have not yet been sent to the terminal. This call is allowed from a background process.

TIOCSTI The argument is a pointer to a **char**. Pretend that character had been received as input.

TIOCGWINSZ The argument is a pointer to a **winsize** structure. The terminal driver's notion of the terminal size is stored into that structure. This call is allowed from a background process.

TIOCSWINSZ The argument is a pointer to a **winsize** structure. The terminal driver's notion of the terminal size is set from the values specified in that structure. If the new sizes are different from the old sizes, a **SIGWINCH** signal is sent to the process group of the terminal.

TIOCMGET The argument is a pointer to an **int**. The current state of the modem status lines is fetched and stored in the **int** pointed to by the argument. This call is allowed from a background process.

TIOCMBIS The argument is a pointer to an **int** whose value is a mask containing modem control lines to be turned on. The control lines whose bits are set in the argument are turned on; no other control lines are affected.

TIOCMBIC The argument is a pointer to an **int** whose value is a mask containing modem control lines to be turned off. The control lines whose bits are set in the argument are turned off; no other control lines are affected.

TIOCMSET The argument is a pointer to an **int** containing a new set of modem control lines. The modem control lines are turned on or off, depending on whether the bit for that mode is set or clear.

TIOCGSOFTCAR The argument is a pointer to an **int** whose value is 1 or 0, depending on whether the software carrier detect is turned on or off.

TIOCSSOFTCAR The argument is a pointer to an **int** whose value is 1 or 0. The value of the integer should be 0 to turn off software carrier, or 1 to turn it on.

SEE ALSO

csh(1), **login**(1), **stty**(1V), **fork**(2V), **getpgrp**(2V), **ioctl**(2), **open**(2V), **read**(2V), **sigvec**(2), **vfork**(2), **tcgetpgrp**(3V), **tty**(4), **ttytab**(5), **getty**(8), **init**(8), **ttysoftcar**(8)

NAME

tfs, TFS – translucent file service

CONFIG

*options***TFS**

SYNOPSIS

#include <sys/mount.h>
mount("tfs", dir, M_NEWTYPE|flags, nfsargs);

DESCRIPTION

The translucent file service (TFS) supplies a copy-on-write filesystem allowing users to share file hierarchies while providing each user with a private hierarchy into which files are copied as they are modified. Consequently, users are isolated from each other's changes.

nfsargs specifies NFS style **mount**(2V) arguments, including the address of the file server (the **tfsd**(8)) and the file handle to be mounted. *dir* is the directory on which the TFS filesystem is to be mounted.

TFS allows a user to mount a private, writable filesystem in front of any number of public, read-only filesystems in such a way that the contents of the public filesystems remain visible behind the contents of the private filesystem. Any change made to a file that is being shared from a public filesystem will cause that file to be copied into the private filesystem, where the modification will be performed.

A directory in a TFS filesystem consists of a number of stacked directories. The searchpath TFS uses to look up a file in a directory corresponds to the stacking order: the TFS will search the "frontmost" directory first, then the directory behind it, and so on until the first occurrence of the file is found. Modifications to a file can be made only in the frontmost directory. TFS copies a file to the frontmost directory when the file is opened for writing with **open**(2V) or when its **stat**(2V) attributes are changed.

If a user removes a file which is not in the frontmost directory, TFS creates a *whiteout* entry in the frontmost directory and leaves the file intact in the back directory. This whiteout entry makes it appear that the file no longer exists, although the file can be reinstated in the directory by using the **unwhiteout**(1) command to remove the whiteout entry. The **lsw**(1) command lists whiteout entries.

TFS filesystems are served by the **tfsd**(8). A TFS filesystem is mounted on a directory by making a TFS_MOUNT protocol request of the **tfsd**, specifying the directories that are to be stacked. The **tfsd** responds with a file handle, which the client then supplies to the **mount**(2V) system call, along with the address of the **tfsd**.

SEE ALSO

lsw(1), **unwhiteout**(1), **mount**(2V), **tfsd**(8), **mount_tfs**(8)

NAME

timod – Transport Interface cooperating STREAMS module

CONFIG

pseudo-device tim64

DESCRIPTION

timod is a STREAMS module for use with the Transport Interface (TI) functions of the Network Services library (see Section 3). The **timod** module converts a set of **ioctl**(2) calls into STREAMS messages that may be consumed by a transport protocol provider which supports the Transport Interface. This allows a user to initiate certain TI functions as atomic operations.

The **timod** module must be pushed onto only a *stream* terminated by a transport protocol provider which supports the TI.

All STREAMS messages, with the exception of the message types generated from the **ioctl**( ) commands described below, are transparently passed to the neighboring STREAMS module or driver. The messages generated from the following **ioctl**( ) commands are recognized and processed by the **timod** module. The format of the **ioctl**( ) call is:

Where, on issuance, **size** is the size of the appropriate TI message to be sent to the transport provider and on return **size** is the size of the appropriate TI message from the transport provider in response to the issued TI message. **buf** is a pointer to a buffer large enough to hold the contents of the appropriate TI messages. The TI message types are defined in **<sys/tihdr.h>**. The possible values for the **cmd** field are:

TI_BIND	Bind an address to the underlying transport protocol provider. The message issued to the TI_BIND **ioctl**( ) is equivalent to the TI message type T_BIND_REQ and the message returned by the successful completion of the **ioctl**( ) is equivalent to the TI message type T_BIND_ACK.
TI_UNBIND	Unbind an address from the underlying transport protocol provider. The message issued to the TI_UNBIND **ioctl**( ) is equivalent to the TI message type T_UNBIND_REQ and the message returned by the successful completion of the **ioctl**( ) is equivalent to the TI message type T_OK_ACK.
TI_GETINFO	Get the TI protocol specific information from the transport protocol provider. The message issued to the TI_GETINFO **ioctl**( ) is equivalent to the TI message type T_INFO_REQ and the message returned by the successful completion of the **ioctl**( ) is equivalent to the TI message type T_INFO_ACK.
TI_OPTMGMT	Get, set or negotiate protocol specific options with the transport protocol provider. The message issued to the TI_OPTMGMT **ioctl**( ) is equivalent to the TI message type T_OPTMGMT_REQ and the message returned by the successful completion of the **ioctl**( ) is equivalent to the TI message type T_OPTMGMT_ACK.

SEE ALSO

tirdwr(4)

Network Programming

DIAGNOSTICS

If the **ioctl**( ) system call returns with a value greater than 0, the lower 8 bits of the return value will be one of the TI error codes as defined in **<sys/tiuser.h>**. If the TI error is of type TSYSERR, then the next 8 bits of the return value will contain an error as defined in **<sys/errno.h>** (see **intro**(2)).

NAME

tirdwr – Transport Interface read/write interface STREAMS module

CONFIG

pseudo-device tirw64

DESCRIPTION

tirdwr is a STREAMS module that provides an alternate interface to a transport provider which supports the Transport Interface (TI) functions of the Network Services library (see Section 3). This alternate interface allows a user to communicate with the transport protocol provider using the **read**(2V) and **write**(2V) system calls. The **putmsg**(2) and **getmsg**(2) system calls may also be used. However, **putmsg**() and **getmsg**() can only transfer data messages between user and *stream*.

The **tirdwr** module must only be pushed (see I_PUSH in **streamio**(4)) onto a **stream** terminated by a transport protocol provider which supports the TI. After the **tirdwr** module has been pushed onto a *stream*, none of the Transport Interface functions can be used. Subsequent calls to TI functions cause an error on the *stream*. Once the error is detected, subsequent system calls on the **stream** return an error with **errno** set to EPROTO.

The following are the actions taken by the **tirdwr** module when pushed on the **stream**, popped (see I_POP in **streamio**(4)) off the *stream*, or when data passes through it.

push When the module is pushed onto a **stream**, it checks any existing data destined for the user to ensure that only regular data messages are present. It ignores any messages on the **stream** that relate to process management, such as messages that generate signals to the user processes associated with the *stream*. If any other messages are present, the I_PUSH returns an error with **errno** set to EPROTO.

write The module takes the following actions on data that originated from a **write**() system call:

All messages with the exception of messages that contain control portions (see **putmsg**(2) and **getmsg**(2)) are transparently passed onto the module's downstream neighbor.

Any zero length data message is freed by the module and is not passed onto the module's downstream neighbor.

Any message with a control portion generates an error, and any further system calls associated with the **stream** fail with **errno** set to EPROTO.

read The module takes the following actions on data that originated from the transport protocol provider:

All messages with the exception of those that contain control portions (see the **putmsg** and **getmsg** system calls) are transparently passed onto the module's upstream neighbor.

The action taken on messages with control portions is as follows:

- Messages that represent expedited data generate an error. All further system calls associated with the **stream** fail with **errno** set to EPROTO.
- Any data messages with control portions have the control portions removed from the message prior to passing the message on to the upstream neighbor.
- Messages that represent an orderly release indication from the transport provider generate a zero length data message, indicating the end of file, which are sent to the reader of the *stream*. The orderly release message itself is freed by the module.
- Messages that represent an abortive disconnect indication from the transport provider cause all further **write**() and **putmsg**() calls to fail with **errno** set to ENXIO. All further **read**() and **getmsg**() calls return zero length data (indicating an EOF) once all previous data has been read.

- With the exception of the above rules, all other messages with control portions generate an error and all further system calls associated with the **stream** fail with **errno** set to EPROTO.

Any zero length data messages are freed by the module and they are not passed onto the module's upstream neighbor.

pop When the module is popped off the **stream** or the **stream** is closed, the module takes the following action:

If an orderly release indication has been previously received, then an orderly release request is sent to the remote side of the transport connection.

SEE ALSO

intro(2), **getmsg**(2), **putmsg**(2), **read**(2V), **write**(2V), **intro**(3), **streamio**(4), **timod**(4)

Network Programming

NAME

tm – Tapemaster 1/2 inch tape controller

CONFIG — SUN-3, SUN-3x SYSTEMS

controller tm0 at vme16d16 ? csr 0xa0 priority 3 vector tmintr 0x60
controller tm1 at vme16d16 ? csr 0xa2 priority 3 vector tmintr 0x61
tape mt0 at tm0 drive 0 flags 1
tape mt0 at tm1 drive 0 flags 1

DESCRIPTION

The Tapemaster tape controller controls Pertec-interface 1/2" tape drives such as the CDC Keystone, providing a standard tape interface to the device, see **mtio**(4). This controller supports single-density or speed drives.

The **tm** driver supports the character device interface. The driver returns an ENOTTY error on unsupported ioctls.

The **tm** driver does not support the backspace file to beginning of file (MTNBSF n) command. The equivalent positioning can be obtained by using MTBSF (n+1) followed by MTFSF 1.

Half-inch reel tape devices do not support the retension ioctl.

FILES

/dev/rmt*	rewinding
/dev/nrmt*	non-rewinding

SEE ALSO

mt(1), **tar**(1), **mtio**(4), **st(4S)**, **xt(4S)**

BUGS

The Tapemaster controller does not provide for byte-swapping and the resultant system overhead prevents streaming transports from streaming.

The system should remember which controlling terminal has the tape drive open and write error messages to that terminal rather than on the console.

The Tapemaster controller is not supported on Sun-4 systems.

WARNINGS

The Tapemaster interface will not be supported in a future release. The Xylogics 472 controller and **xt** driver replace the Tapemaster controller and **tm** driver.

NAME

tmpfs – memory based filesystem

CONFIG

options TMPFS

SYNOPSIS

#include <sys/mount.h>
mount ("tmpfs", dir, M_NEWTYPE | flags, args);

DESCRIPTION

tmpfs is a memory based filesystem which uses kernel resources relating to the VM system and page cache as a filesystem. Once mounted, a **tmpfs** filesystem provides standard file operations and semantics. **tmpfs** is so named because files and directories are not preserved across reboot or unmounts, all files residing on a **tmpfs** filesystem that is unmounted will be lost.

tmpfs filesystems are mounted either with the command:

mount –t tmp swap *directory-name*

or by placing the line

swap *directory-name* **tmp rw 0 0**

in your **/etc/fstab** file and using the **mount**(8) command as normal. The **/etc/rc.local** file contains commands to mount a **tmpfs** filesystem on **/tmp** at multi-user startup time but is by default commented out. To mount a **tmpfs** filesystem on **/tmp** (maximizing possible performance improvements), add the above line to **/etc/fstab** and uncomment the following line in **/etc/rc.local**:

#mount /tmp

tmpfs is designed as a performance enhancement which is achieved by cacheing the writes to files residing on a **tmpfs** filesystem. Performance improvements are most noticeable when a large number of short lived files are written and accessed on a **tmpfs** filesystem. Large compilations with **tmpfs** mounted on **/tmp** are a good example of this.

Users of **tmpfs** should be aware of some tradeoffs involved in mounting a **tmpfs** filesystem. The resources used by **tmpfs** are the same as those used when commands are executed (for example, swap space allocation). This means that a large sized or number of **tmpfs** files can affect the amount of space left over for programs to execute. Likewise, programs requiring large amounts of memory use up the space available to **tmpfs**. Users running into these constraints (for example, running out of space on **tmpfs**) can allocate more swap space by using the **swapon**(8) command.

Normal filesystem writes are scheduled to be written to a permanent storage medium along with all control information associated with the file (for example, modification time, file permissions). **tmpfs** control information resides only in memory and never needs to be written to permanent storage. File data remains in core until memory demands are sufficient to cause pages associated with **tmpfs** to be reused at which time they are copied out to swap.

SEE ALSO

df(1V), **mount**(2V), **umount**(2V), **fstab**(5), **mount**(8), **swapon**(8)

System Services Overview,
System and Network Administration

NOTES

swapon to a **tmpfs** file is not supported.

df(1V) output is of limited accuracy since a **tmpfs** filesystem size is not static and the space available to **tmpfs** is dependent on the swap space demands of the entire system.

DIAGNOSTICS

If **tmpfs** runs out of space, one of the following messages will be printed to the console.

directory: **file system full, anon reservation exceeded**

directory: **file system full, anon allocation exceeded**

A page could not be allocated while writing to a file. This can occur if **tmpfs** is attempting to write more than it is allowed, or if currently executing programs are using a lot of memory. To make more space available, remove unneccessary files, exit from some programs, or allocate more swap space using **swapon**(8).

directory: **file system full, kmem_alloc failure**

tmpfs ran out of physical memory while attempting to create a new file or directory. Remove unneccesary files or directories or install more physical memory.

WARNINGS

A **tmpfs** filesystem should *not* be mounted on **/var/tmp**, this directory is used by vi(1) for preserved files. Files and directories on a **tmpfs** filesystem are not preserved across reboots or unmounts. Command scripts or programs which count on this will not work as expected.

NAME

ttcompat – V7 and 4BSD STREAMS compatibility module

CONFIG

None; included by default.

SYNOPSIS

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

ioctl(fd, I_PUSH, "ttcompat");
```

DESCRIPTION

ttcompat is a STREAMS module that translates the **ioctl** calls supported by the older Version 7 and 4BSD terminal drivers into the **ioctl** calls supported by the **termio**(4) interface. All other messages pass through this module unchanged; the behavior of **read** and **write** calls is unchanged, as is the behavior of **ioctl** calls other than the ones supported by **ttcompat**.

Normally, this module is automatically pushed onto a stream when a terminal device is opened; it does not have to be explicitly pushed onto a stream. This module requires that the **termio** interface be supported by the modules and driver downstream. The TCGETS, TCSETS, and TCSETSF **ioctl** calls must be supported; if any information set or fetched by those **ioctl** calls is not supported by the modules and driver downstream, some of the V7/4BSD functions may not be supported. For example, if the CBAUD bits in the **c_cflag** field are not supported, the functions provided by the **sg_ispeed** and **sg_ospeed** fields of the **sgttyb** structure (see below) will not be supported. If the TCFLSH **ioctl** is not supported, the function provided by the TIOCFLUSH **ioctl** will not be supported. If the TCXONC **ioctl** is not supported, the functions provided by the TIOCSTOP and TIOCSTART **ioctl** calls will not be supported. If the TIOCMBIS and TIOCMBIC **ioctl** calls are not supported, the functions provided by the TIOCSDTR and TIOCCDTR **ioctl** calls will not be supported.

The basic **ioctl** calls use the **sgttyb** structure defined by **<sys/ioctl.h>**:

```
struct sgttyb {
        char    sg_ispeed;
        char    sg_ospeed;
        char    sg_erase;
        char    sg_kill;
        short   sg_flags;
};
```

The **sg_ispeed** and **sg_ospeed** fields describe the input and output speeds of the device, and reflect the values in the **c_cflag** field of the **termio** structure. The **sg_erase** and **sg_kill** fields of the argument structure specify the erase and kill characters respectively, and reflect the values in the VERASE and VKILL members of the **c_cc** field of the **termio** structure.

The **sg_flags** field of the argument structure contains several flags that determine the system's treatment of the terminal. They are mapped into flags in fields of the terminal state, represented by the **termio** structure.

Delay type 0 is always mapped into the equivalent delay type 0 in the **c_oflag** field of the **termio** structure. Other delay mappings are performed as follows:

sg_flags	c_oflag
BS1	BS1
FF1	VT1
CR1	CR2
CR2	CR3
CR3	not supported
TAB1	TAB1
TAB2	TAB2

XTABS	TAB3
NL1	ONLRET|CR1
NL2	NL1

If previous TIOCLSET or TIOCLBIS ioctl calls have not selected LITOUT or PASS8 mode, and if RAW mode is not selected, the ISTRIP flag is set in the c_iflag field of the termio structure, and the EVENP and ODDP flags control the parity of characters sent to the terminal and accepted from the terminal:

0 Parity is not to be generated on output or checked on input; the character size is set to CS8 and the PARENB flag is cleared in the c_cflag field of the termio structure.

EVENP Even parity characters are to be generated on output and accepted on input; the INPCK flag is set in the c_iflag field of the termio structure, the character size is set to CS7 and the PARENB flag is set in the c_cflag field of the termio structure.

ODDP Odd parity characters are to be generated on output and accepted on input; the INPCK flag is set in the c_iflag field, the character size is set to CS7 and the PARENB and PARODD flags are set in the c_cflag field of the termio structure.

EVENP|ODDP
Even parity characters are to be generated on output and characters of either parity are to be accepted on input; the INPCK flag is cleared in the c_iflag field, the character size is set to CS7 and the PARENB flag is set in the c_cflag field of the termio structure.

The RAW flag disables all output processing (the OPOST flag in the c_oflag field, and the XCASE flag in the c_lflag field, are cleared in the termio structure) and input processing (all flags in the c_iflag field other than the IXOFF and IXANY flags are cleared in the termio structure). 8 bits of data, with no parity bit, are accepted on input and generated on output; the character size is set to CS8 and the PARENB and PARODD flags are cleared in the c_cflag field of the termio structure. The signal-generating and line-editing control characters are disabled by clearing the ISIG and ICANON flags in the c_lflag field of the termio structure.

The CRMOD flag turn input RETURN characters into NEWLINE characters, and output and echoed NEWLINE characters to be output as a RETURN followed by a LINEFEED. The ICRNL flag in the c_iflag field, and the OPOST and ONLCR flags in the c_oflag field, are set in the termio structure.

The LCASE flag maps upper-case letters in the ASCII character set to their lower-case equivalents on input (the IUCLC flag is set in the c_iflag field), and maps lower-case letters in the ASCII character set to their upper-case equivalents on output (the OLCUC flag is set in the c_oflag field). Escape sequences are accepted on input, and generated on output, to handle certain ASCII characters not supported by older terminals (the XCASE flag is set in the c_lflag field).

Other flags are directly mapped to flags in the termio structure:

sg_flags	flags in termio structure
CBREAK	complement of ICANON in c_lflag field
ECHO	ECHO in c_lflag field
TANDEM	IXOFF in c_iflag field

Another structure associated with each terminal specifies characters that are special in both the old Version 7 and the newer 4BSD terminal interfaces. The following structure is defined by <sys/ioctl.h>:

```
struct tchars {
        char    t_intrc;        /* interrupt */
        char    t_quitc;        /* quit */
        char    t_startc;       /* start output */
        char    t_stopc;        /* stop output */
        char    t_eofc;         /* end-of-file */
        char    t_brkc;         /* input delimiter (like nl) */
};
```

The characters are mapped to members of the **c_cc** field of the **termio** structure as follows:

tchars	**c_cc** index
t_intrc	**VINTR**
t_quitc	**VQUIT**
t_startc	**VSTART**
t_stopc	**VSTOP**
t_eofc	**VEOF**
t_brkc	**VEOL**

Also associated with each terminal is a local flag word, specifying flags supported by the new 4BSD terminal interface. Most of these flags are directly mapped to flags in the **termio** structure:

local flags	flags in **termio** structure
LCRTBS	not supported
LPRTERA	**ECHOPRT** in the **c_lflag** field
LCRTERA	**ECHOE** in the **c_lflag** field
LTILDE	not supported
LTOSTOP	**TOSTOP** in the **c_lflag** field
LFLUSHO	**FLUSHO** in the **c_lflag** field
LNOHANG	**CLOCAL** in the **c_cflag** field
LCRTKIL	**ECHOKE** in the **c_lflag** field
LCTLECH	**CTLECH** in the **c_lflag** field
LPENDIN	**PENDIN** in the **c_lflag** field
LDECCTQ	complement of **IXANY** in the **c_iflag** field
LNOFLSH	**NOFLSH** in the **c_lflag** field

Another structure associated with each terminal is the **ltchars** structure which defines control characters for the new 4BSD terminal interface. Its structure is:

```
struct ltchars {
        char    t_suspc;        /* stop process signal */
        char    t_dsuspc;       /* delayed stop process signal */
        char    t_rprntc;       /* reprint line */
        char    t_flushc;       /* flush output (toggles) */
        char    t_werasc;       /* word erase */
        char    t_lnextc;       /* literal next character */
};
```

The characters are mapped to members of the **c_cc** field of the **termio** structure as follows:

ltchars	**c_cc** index
t_suspc	**VSUSP**
t_dsuspc	**VDSUSP**
t_rprntc	**VREPRINT**
t_flushc	**VDISCARD**
t_werasc	**VWERASE**
t_lnextc	**VLNEXT**

IOCTLS

ttcompat responds to the following **ioctl** calls. All others are passed to the module below.

TIOCGETP The argument is a pointer to an **sgttyb** structure. The current terminal state is fetched; the appropriate characters in the terminal state are stored in that structure, as are the input and output speeds. The values of the flags in the **sg_flags** field are derived from the flags in the terminal state and stored in the structure.

TIOCSETP The argument is a pointer to an **sgttyb** structure. The appropriate characters and input and output speeds in the terminal state are set from the values in that structure, and the flags in the terminal state are set to match the values of the flags in the **sg_flags** field of that structure. The state is changed with a **TCSETSF** *ioctl*, so that the interface delays until output is quiescent, then throws away any unread characters, before changing the modes.

TIOCSETN The argument is a pointer to an **sgttyb** structure. The terminal state is changed as **TIOCSETP** would change it, but a **TCSETS ioctl** is used, so that the interface neither delays nor discards input.

TIOCHPCL The argument is ignored. The **HUPCL** flag is set in the **c_cflag** word of the terminal state.

TIOCFLUSH The argument is a pointer to an **int** variable. If its value is zero, all characters waiting in input or output queues are flushed. 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** bit is set, all characters waiting in input queues are flushed, and if the **FWRITE** bit is set, all characters waiting in output queues are flushed.

TIOCSBRK The argument is ignored. The break bit is set for the device.

TIOCCBRK The argument is ignored. The break bit is cleared for the device.

TIOCSDTR The argument is ignored. The Data Terminal Ready bit is set for the device.

TIOCCDTR The argument is ignored. The Data Terminal Ready bit is cleared for the device.

TIOCSTOP The argument is ignored. Output is stopped as if the **STOP** character had been typed.

TIOCSTART The argument is ignored. Output is restarted as if the **START** character had been typed.

TIOCGETC The argument is a pointer to an **tchars** structure. The current terminal state is fetched, and the appropriate characters in the terminal state are stored in that structure.

TIOCSETC The argument is a pointer to an **tchars** structure. The values of the appropriate characters in the terminal state are set from the characters in that structure.

TIOCLGET The argument is a pointer to an **int**. The current terminal state is fetched, and the values of the local flags are derived from the flags in the terminal state and stored in the **int** pointed to by the argument.

TIOCLBIS The argument is a pointer to an **int** whose value is a mask containing flags to be set in the local flags word. The current terminal state is fetched, and the values of the local flags are derived from the flags in the terminal state; the specified flags are set, and the flags in the terminal state are set to match the new value of the local flags word.

TIOCLBIC The argument is a pointer to an **int** whose value is a mask containing flags to be cleared in the local flags word. The current terminal state is fetched, and the values of the local flags are derived from the flags in the terminal state; the specified flags are cleared, and the flags in the terminal state are set to match the new value of the local flags word.

TIOCLSET The argument is a pointer to an **int** containing a new set of local flags. The flags in the terminal state are set to match the new value of the local flags word.

TIOCGLTC The argument is a pointer to an **ltchars** structure. The values of the appropriate characters in the terminal state are stored in that structure.

TIOCSLTC The argument is a pointer to an **ltchars** structure. The values of the appropriate characters in the terminal state are set from the characters in that structure.

SEE ALSO

ioctl(2), **termio**(4)

NAME

tty – controlling terminal interface

DESCRIPTION

The file **/dev/tty** is, in each process, a synonym for the controlling terminal of that process, if any. It is useful for programs or shell sequences that wish 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.

IOCTLS

In addition to the **ioctl**( ) requests supported by the device that **tty** refers to, the following **ioctl**( ) request is supported:

TIOCNOTTY Detach the current process from its controlling terminal, and remove it from its current process group, without attaching it to a new process group (that is, set its process group ID to zero). This **ioctl**( ) call only works on file descriptors connected to **/dev/tty**; this is used by daemon processes when they are invoked by a user at a terminal. The process attempts to open **/dev/tty**; if the open succeeds, it detaches itself from the terminal by using TIOCNOTTY, while if the open fails, it is obviously not attached to a terminal and does not need to detach itself.

FILES

/dev/tty

SEE ALSO

termio(4)

NAME

udp – Internet User Datagram Protocol

SYNOPSIS

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

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

DESCRIPTION

UDP is a simple, unreliable datagram protocol which is used to support the SOCK_DGRAM abstraction for the Internet protocol family. It is layered directly above the Internet Protocol (IP). UDP sockets are connectionless, and are normally used with the **sendto**, **sendmsg**, **recvfrom**, and **recvmsg** system calls (see **send**(2) and **recv**(2)). If the **connect**(2) system call is used to fix the destination for future packets, then the **recv**(2) or **read**(2V) and **send**(2) or **write**(2V) system calls may be used.

UDP address formats are identical to those used by the Transmission Control Protocol (TCP). Like TCP, UDP uses a port number along with an IP address to identify the endpoint of communication. Note: the UDP port number space is separate from the TCP port number space (that is, a UDP port may not be "connected" to a TCP port). The **bind**(2) system call can be used to set the local address and port number of a UDP socket. The local IP address may be left unspecified in the **bind** call by using the special value INADDR_ANY. If the **bind** call is not done, a local IP address and port number will be assigned to each packet as it is sent. Broadcast packets may be sent (assuming the underlying network supports this) by using a reserved "broadcast address"; this address is network interface dependent. Broadcasts may only be sent by the super-user.

Options at the IP level may be used with UDP; see **ip**(4P).

There are a variety of ways that a UDP packet can be lost or discarded, including a failure of the underlying communication mechanism. UDP implements a checksum over the data portion of the packet. If the checksum of a received packet is in error, the packet will be dropped with no indication given to the user. A queue of received packets is provided for each UDP socket. This queue has a limited capacity. Arriving datagrams which will not fit within its *high-water* capacity are silently discarded.

UDP processes Internet Control Message Protocol (ICMP) error messages received in response to UDP packets it has sent. See **icmp**(4P). ICMP "source quench" messages are ignored. ICMP "destination unreachable," "time exceeded" and "parameter problem" messages disconnect the socket from its peer so that subsequent attempts to send packets using that socket will return an error. UDP will not guarantee that packets are delivered in the order they were sent. As well, duplicate packets may be generated in the communication process.

ERRORS

A socket operation may fail if:

EISCONN	A **connect** operation was attempted on a socket on which a **connect** operation had already been performed, and the socket could not be successfully disconnected before making the new connection.
EISCONN	A **sendto** or **sendmsg** operation specifying an address to which the message should be sent was attempted on a socket on which a **connect** operation had already been performed.
ENOTCONN	A **send** or **write** operation, or a **sendto** or **sendmsg** operation not specifying an address to which the message should be sent, was attempted on a socket on which a **connect** operation had not already been performed.
EADDRINUSE	A **bind** operation was attempted on a socket with a network address/port pair that has already been bound to another socket.
EADDRNOTAVAIL	A **bind** operation was attempted on a socket with a network address for which no network interface exists.

EINVAL	A **sendmsg** operation with a non-NULL **msg_accrights** was attempted.
EACCES	A **bind** operation was attempted with a "reserved" port number and the effective user ID of the process was not super-user.
ENOBUFS	The system ran out of memory for internal data structures.

SEE ALSO

bind(2), **connect**(2), **read**(2V), **recv**(2), **send**(2), **write**(2V), **icmp**(4P), **inet**(4F), **ip**(4P), **tcp**(4P)

Postel, Jon, *User Datagram Protocol*, RFC 768, Network Information Center, SRI International, Menlo Park, Calif., August 1980. (Sun 800-1054-01)

BUGS

SIOCSHIWAT and **SIOCGHIWAT** ioctl's to set and get the high water mark for the socket queue, and so that it can be changed from 2048 bytes to be larger or smaller, have been defined (in **sys/ioctl.h**) but not implemented.

Something sensible should be done with ICMP source quench error messages if the socket is bound to a peer socket.

NAME

unix – UNIX domain protocol family

DESCRIPTION

The Unix Domain protocol family provides support for socket-based communication between processes running on the local host. While both **SOCK_STREAM** and **SOCK_DGRAM** types are supported, the **SOCK_STREAM** type often provides faster performance. Pipes, for instance, are built on Unix Domain **SOCK_STREAM** sockets.

Unix Domain **SOCK_DGRAM** sockets (also called datagram sockets) exist primarily for reasons of orthogonality under the BSD socket model. However, the overhead of reading or writing data is higher for the (connectionless) datagram sockets.

Unix Domain addresses are pathnames. In other words, two independent processes can communicate by specifying the same pathname as their communications rendezvous point. The **bind**(2) operation creates a special entry in the file system of type socket. If that pathname already exists (as a socket from a previous **bind**() operation, or as some other file system type), **bind**() will fail.

Sockets in the Unix domain protocol family use the following addressing structure:

```
struct sockaddr_un {
        short   sun_family;
        u_short sun_path[108];
};
```

To create or reference a Unix Domain socket, the **sun_family** field should be set to **AF_UNIX** and the **sun_path** array should contain the path name of a rendezvous point.

Although Unix Domain sockets are faster than Internet Domain sockets for communication between local processes, the advantage of the additional flexibility afforded by the latter may outweigh performance issues. Where inter-process communication thoughput is critical, a shared memory approach may be preferred.

Since there are no protocol families associated with Unix Domain sockets, the protocol argument to **socket**(2) should be zero.

When setting up a Unix Domain socket, the *length* argument to the **bind**() call is the amount of space within the **sockaddr_un** structure, not including the pathname delimiter. One way to specify the length is:

sizeof(*addr.sun_family*) + **strlen**(*path*) where *addr* is a structure of type **sockaddr_un**, and *path* is a pointer to the pathname.

The limit of 108 characters is an artifact of the implementation.

Since closing a Unix Domain socket does not make the file system entry go away, an application should remove the entry using **unlink**(2V), when finished.

SEE ALSO

bind(2), **socket**(2), **unlink**(2V)

Network Programming

NAME

vd – loadable modules interface

CONFIG

None; included with **options VDDRV**

DESCRIPTION

This pseudo-device provides kernel support for loadable modules. It is used exclusively by the **modload**(8), **modunload**(8), and **modstat**(8) utilities. Other programs should not use it.

FILES

/dev/vd

SEE ALSO

modload(), **modunload**(), **modstat**()

WARNINGS

The interface provided by **vd** is subject to change without notice.

NAME

vpc – Systech VPC-2200 Versatec printer/plotter and Centronics printer interface

CONFIG

device vpc0 at vme16d16 ? csr 0x480 priority 2 vector vpcintr 0x80
device vpc1 at vme16d16 ? csr 0x500 priority 2 vector vpcintr 0x81

AVAILABILITY

Sun-3, Sun-3/80 and Sun-4 systems only.

DESCRIPTION

This Sun interface to the Versatec printer/plotter and to Centronics printers is supported by the Systech parallel interface board, an output-only byte-wide DMA device. The device has one channel for Versatec devices and one channel for Centronics devices, with an optional long lines interface for Versatec devices.

Devices attached to this interface are normally handled by the line printer spooling system and should not be accessed by the user directly.

Opening the device **/dev/vpc0** or **/dev/lp0** may yield one of two errors: ENXIO indicates that the device is already in use; EIO indicates that the device is offline.

The Versatec printer/plotter operates in either print or plot mode. To set the printer into plot mode you should include **<sys/vcmd.h>** and use the **ioctl**(2) call:

```
ioctl(f, VSETSTATE, plotmd);
```

where *plotmd* is defined to be

```
int plotmd[ ] = { VPLOT, 0, 0 };
```

When going back into print mode from plot mode you normally eject paper by sending it an EOT after putting into print mode:

```
int prtmd[ ] = { VPRINT, 0, 0 };
...
fflush (vpc);
f = fileno(vpc);
ioctl(f, VSETSTATE, prtmd);
write(f, "\04", 1);
```

FILES

/dev/vpc0
/dev/lp0

SEE ALSO

ioctl(2), **setbuf**(3V)

BUGS

If you use the standard I/O library on the Versatec, be sure to explicitly set a buffer using **setbuf**(3V), since the library will not use buffered output by default, and will run very slowly.

NAME

win – Sun window system

CONFIG

pseudo-device win*number*
pseudo-device dtop*number*

DESCRIPTION

The **win** pseudo-device accesses the system drivers supporting the Sun window system. *number*, in the device description line above, indicates the maximum number of windows supported by the system. *number* is set to 128 in the GENERIC system configuration file used to generate the kernel used in Sun systems as they are shipped. The *dtop* pseudo-device line indicates the number of separate "desktops" (frame buffers) that can be actively running the Sun window system at once. In the GENERIC file, this number is set to 4.

Each window in the system is represented by a **/dev/win*** device. The windows are organized as a tree with windows being subwindows of their parents, and covering/covered by their siblings. Each window has a position in the tree, a position on a display screen, an input queue, and information telling what parts of it are exposed.

The window driver multiplexes keyboard and mouse input among the several windows, tracks the mouse with a cursor on the screen, provides each window access to information about what parts of it are exposed, and notifies the manager process for a window when the exposed area of the window changes so that the window may repair its display.

Full information on the window system functions is given in the *SunView System Programmer's Guide*.

FILES

/dev/win[0-9]
/dev/win[0-9][0-9]

SEE ALSO

SunView System Programmer's Guide

NAME

xd – Disk driver for Xylogics 7053 SMD Disk Controller

CONFIG — SUN-3, SUN-3x, SUN-4 SYSTEMS

controller xdc0 at vme16d32 ? csr 0xee80 priority 2 vector xdintr 0x44
controller xdc1 at vme16d32 ? csr 0xee90 priority 2 vector xdintr 0x45
controller xdc2 at vme16d32 ? csr 0xeea0 priority 2 vector xdintr 0x46
controller xdc3 at vme16d32 ? csr 0xeeb0 priority 2 vector xdintr 0x47
disk xd0 at xdc0 drive 0
disk xd1 at xdc0 drive 1
disk xd2 at xdc0 drive 2
disk xd3 at xdc0 drive 3
disk xd4 at xdc1 drive 0
disk xd5 at xdc1 drive 1
disk xd6 at xdc1 drive 2
disk xd7 at xdc1 drive 3
disk xd8 at xdc2 drive 0
disk xd9 at xdc2 drive 1
disk xd10 at xdc2 drive 2
disk xd11 at xdc2 drive 3
disk xd12 at xdc3 drive 0
disk xd13 at xdc3 drive 1
disk xd14 at xdc3 drive 2
disk xd15 at xdc3 drive 3

The four **controller** lines given in the synopsis section above specify the first, second, third, and fourth Xylogics 7053 SMD disk controller in a Sun system.

DESCRIPTION

Files with minor device numbers 0 through 7 refer to various portions of drive 0; minor devices 8 through 15 refer to drive 1, and so on. The standard device names begin with **xd** followed by the drive number and then a letter a-h for partitions 0-7 respectively. The character ? stands here for a drive number in the range 0-7.

The block files access the disk using the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a "raw" interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call usually results in only one I/O operation; therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw files conventionally begin with an extra **r**.

In raw I/O counts should be a multiple of 512 bytes (a disk sector). Likewise **directory**(3V) calls should specify a multiple of 512 bytes.

If **flags 0x1** is specified, the overlapped seeks feature for that drive is turned off. Note: to be effective, the flag must be set on all drives for a specific controller. This action is necessary for controllers with older firmware, which have bugs preventing overlapped seeks from working properly.

DISK SUPPORT

This driver handles all SMD drives by reading a label from sector 0 of the drive which describes the disk geometry and partitioning.

The **xd?a** partition is normally used for the root file system on a disk, the xd?b partition as a paging area, and the **xd?c** partition for pack-pack copying (it normally maps the entire disk). The rest of the disk is normally the **xd?g** partition.

FILES

/dev/xd[0-7][a-h]	block files
/dev/rxd[0-7][a-h]	raw files

SEE ALSO

lseek(2V), **read**(2V), **write**(2V), **directory**(3V), **dkio**(4S)

DIAGNOSTICS

xdc*n*: self test error

Self test error in controller, see the Maintenance and Reference Manual.

xd*n*: unable to read bad sector

The bad sector forwarding information for the disk could not be read.

xd*n*: initialization failed

The drive could not be successfully initialized.

xd*n*: unable to read label

The drive geometry/partition table information could not be read.

xd*n*: Corrupt label

The geometry/partition label checksum was incorrect.

xd*n*: offline

A drive ready status is no longer detected, so the unit has been logically removed from the system. If the drive ready status is restored, the unit will automatically come back online the next time it is accessed.

xd*nc*: *cmd how* (*msg*) **blk** *#n abs blk #n*

A command such as read or write encountered an error condition (*how*): either it *failed*, the controller was *reset*, the unit was *restored*, or an operation was *retry*'ed. The *msg* is derived from the error number given by the controller, indicating a condition such as "drive not ready(rq, "sector not found" or "disk write protected". The *blk #* is the sector in error relative to the beginning of the partition involved. The *abs blk #* is the absolute block number of the sector in error. Some fields of the error message may be missing since the information is not always available.

BUGS

In raw I/O **read**(2V) and **write**(2V) truncate file offsets to 512-byte block boundaries, and **write**(2V) scribbles on the tail of incomplete blocks. Thus, in programs that are likely to access raw devices, **read**(2V), **write**(2V) and **lseek**(2V) should always deal in 512-byte multiples.

Older revisions of the firmware do not properly support overlapped seeks. This will only affect systems with multiple disks on a single controller. If a large number of "zero sector count" errors appear, you should use the **flags** field to disable overlapped seeks.

NAME

xt – Xylogics 472 1/2 inch tape controller

CONFIG — SUN-3, SUN-4 SYSTEMS

controller xtc0 at vme16d16 ? csr 0xee60 priority 3 vector xtintr 0x64
controller xtc1 at vme16d16 ? csr 0xee68 priority 3 vector xtintr 0x65
tape xt0 at xtc0 drive 0 flags 1
tape xt1 at xtc1 drive 0 flags 1

DESCRIPTION

The Xylogics 472 tape controller controls Pertec-interface 1/2" tape drives such as the Fujitsu M2444 and the CDC Keystone III, providing a standard tape interface to the device see **mtio**(4). This controller is used to support high speed or high density drives, which are not supported effectively by the older Tapemaster controller (see **tm**(4S)).

The flags field is used to control remote density select operation: a 0 specifies no remote density selection is to be attempted, a 1 specifies that the Pertec density-select line is used to toggle between high and low density; a 2 specifies that the Pertec speed-select line is used to toggle between high and low density. The default is 1, which is appropriate for the Fujitsu M2444, the CDC Keystone III (92185) and the Telex 9250. In no case will the controller select among more than 2 densities.

The **xt** driver supports the character device interface.

EOT Handling

The user will be notified of end of tape (EOT) on write by a 0 byte count returned the first time this is attempted. This write must be retried by the user. Subsequent writes will be successful until the tape winds off the reel. Read past EOT is transparent to the user.

Ioctls

Not all devices support all ioctls. The driver returns an ENOTTY error on unsupported ioctls.

1/2" tape devices do not support the tape retension function.

FILES

/dev/rmt0	low density operation, typically 1600 bpi
/dev/rmt8	high density operation, typically 6250 bpi
/dev/nrmt*	non-rewinding

SEE ALSO

mt(1), **tar**(1), **mtio**(4), **st**(4S), **suninstall**(8)

BUGS

Record sizes are restricted to an even number of bytes.

Absolute file positioning is not fully supported; it is only meant to be used by **suninstall**(8).

NAME

xy – Disk driver for Xylogics 450 and 451 SMD Disk Controllers

CONFIG — SUN-3, SUN-3x, SUN-4 SYSTEMS

controller xyc0 at vme16d16 ? csr 0xee40 priority 2 vector xyintr 0x48
controller xyc1 at vme16d16 ? csr 0xee48 priority 2 vector xyintr 0x49
disk xy0 at xyc0 drive 0
disk xy1 at xyc0 drive 1
disk xy2 at xyc1 drive 0
disk xy3 at xyc1 drive 1

The two **controller** lines given in the synopsis sections above specify the first and second Xylogics 450 or 451 SMD disk controller in a Sun system.

DESCRIPTION

Files with minor device numbers 0 through 7 refer to various portions of drive 0; minor devices 8 through 15 refer to drive 1, and so on. The standard device names begin with **xy** followed by the drive number and then a letter a-h for partitions 0-7 respectively. The character '**?**' stands here for a drive number in the range 0-7.

The block files access the disk using the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a "raw" interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call usually results in only one I/O operation; therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw files conventionally begin with an extra **r**.

When using raw I/O, transfer counts should be multiples of 512 bytes (the size of a disk sector). Likewise, when using **lseek**(2V) to specify block offsets from which to perform raw I/O, the logical offset should also be a multiple of 512 bytes.

Due to word ordering differences between the disk controller and Sun computers, user buffers that are used for raw I/O must not begin on odd byte boundaries.

If **flags 0x1** is specified, the overlapped seeks feature for that drive is turned off. Note: to be effective, the flag must be set on all drives for a specific controller. This action is necessary for controllers with older firmware, which have bugs preventing overlapped seeks from working properly.

DISK SUPPORT

This driver handles all SMD drives by reading a label from sector 0 of the drive which describes the disk geometry and partitioning.

The **xy?a** partition is normally used for the root file system on a disk, the **xy?b** partition as a paging area, and the **xy?c** partition for pack-pack copying (it normally maps the entire disk). The rest of the disk is normally the **xy?g** partition.

FILES

/dev/xy[0-7][a-h]	block files
/dev/rxy[0-7][a-h]	raw files

SEE ALSO

lseek(2V), **read**(2V), **directory**(3V), **write**(2V), **dkio**(4S)

DIAGNOSTICS

xyc*n* : self test error
Self test error in controller, see the Maintenance and Reference Manual.

xyc*n*: WARNING: *n* bit addresses
The controller is strapped incorrectly. Sun systems use 20-bit addresses for Multibus based systems and 24-bit addresses for VMEbus based systems.

xy*n* : unable to read bad sector info
The bad sector forwarding information for the disk could not be read.

xy*n* and xy*n* are of same type (*n*) with different geometries.
The 450 and 451 do not support mixing the drive types found on these units on a single controller.

xy*n* : initialization failed
The drive could not be successfully initialized.

xy*n* : unable to read label
The drive geometry/partition table information could not be read.

xy*n* : Corrupt label
The geometry/partition label checksum was incorrect.

xy*n* : offline
A drive ready status is no longer detected, so the unit has been logically removed from the system. If the drive ready status is restored, the unit will automatically come back online the next time it is accessed.

xy*nc*: *cmd how* (*msg*) **blk** *#n abs blk #n*
A command such as read or write encountered an error condition (*how*): either it *failed*, the controller was *reset*, the unit was *restored*, or an operation was *retry*'ed. The *msg* is derived from the error number given by the controller, indicating a condition such as "drive not ready", "sector not found" or "disk write protected". The *blk #* is the sector in error relative to the beginning of the partition involved. The *abs blk #* is the absolute block number of the sector in error. Some fields of the error message may be missing since the information is not always available.

BUGS

In raw I/O **read**(2V) and **write**(2V) truncate file offsets to 512-byte block boundaries, and **write**(2V) scribbles on the tail of incomplete blocks. Thus, in programs that are likely to access raw devices, **read**(2V), **write**(2V) and **lseek**(2V) should always deal in 512-byte multiples.

Older revisions of the firmware do not properly support overlapped seeks. This will only affect systems with multiple disks on a single controller. If a large number of "zero sector count" errors appear, you should use the **flags** field to disable overlapped seeks.

NAME

zero – source of zeroes

SYNOPSIS

None; included with standard system.

DESCRIPTION

A zero special file is a source of zeroed unnamed memory.

Reads from a zero special file always return a buffer full of zeroes. The file is of infinite length.

Writes to a zero special file are always successful, but the data written is ignored.

Mapping a zero special file creates a zero-initialized unnamed memory object of a length equal to the length of the mapping and rounded up to the nearest page size as returned by **getpagesize**(2). Multiple processes can share such a zero special file object provided a common ancestor mapped the object **MAP_SHARED**.

FILES

/dev/zero

SEE ALSO

fork(2V), **getpagesize**(2), **mmap**(2)

NAME

zs – Zilog 8530 SCC serial communications driver

CONFIG — SUN-3 SYSTEM

device zs0 at obio ? csr 0x20000 flags 3 priority 3
device zs1 at obio ? csr 0x00000 flags 0x103 priority 3

CONFIG — SUN-3x SYSTEM

device zs0 at obio ? csr 0x62002000 flags 3 priority 3
device zs1 at obio ? csr 0x62000000 flags 0x103 priority 3

CONFIG — SUN-4 SYSTEM

device zs1 at obio ? csr 0xf0000000 flags 0x103 priority 3
device zs2 at obio 3 csr 0xe0000000 flags 3 priority 3

CONFIG — SPARCSTATION 1 SYSTEM

device-driver zs

CONFIG — Sun386i SYSTEM

device zs0 at obmem ? csr 0xFC000000 flags 3 irq 9 priority 6
device zs1 at obmem ? csr 0xA0000020 flags 0x103 irq 9 priority 6

SYNOPSIS

#include <fcntl.h>
#include <sys/termios.h>
open("/dev/tty*n*", mode);
open("/dev/ttyd*n*", mode);
open("/dev/cua*n*", mode);

DESCRIPTION

The Zilog 8530 provides 2 serial communication ports with full modem control in asynchronous mode. Each port supports those **termio**(4) device control functions specified by flags in the **c_cflag** word of the **termios** structure and by the **IGNBRK, IGNPAR, PARMRK,** or **INPCK** flags in the **c_iflag** word of the **termios** structure are performed by the **zs** driver. All other **termio**(4) functions must be performed by STREAMS modules pushed atop the driver; when a device is opened, the **ldterm**(4M) and **ttcompat**(4M) STREAMS modules are automatically pushed on top of the stream, providing the standard **termio**(4) interface.

Of the synopsis lines above, the line for **zs0** specifies the serial I/O port(s) provided by the CPU board, the line for **zs1** specifies the Video Board ports (which are used for keyboard and mouse), the lines for **zs2** and **zs3** specify the first and second ports on the first SCSI board in a system, and those for **zs4** and **zs5** specify the first and second ports provided by the second SCSI board in a system, respectively.

Bit *i* of **flags** may be specified to say that a line is not properly connected, and that the line *i* should be treated as hard-wired with carrier always present. Thus specifying **flags 0x2** in the specification of **zs0** would treat line **/dev/ttyb** in this way.

Minor device numbers in the range 0 – 11 correspond directly to the normal tty lines and are named **/dev/ttya** and **/dev/ttyb** for the two serial ports on the CPU board and **/dev/ttys***n* for the ports on the SCSI boards; *n* is 0 or 1 for the ports on the first SCSI board, and 2 or 3 for the ports on the second SCSI board.

To allow a single tty line to be connected to a modem and used for both incoming and outgoing calls, a special feature, controlled by the minor device number, has been added. Minor device numbers in the range 128 – 139 correspond to the same physical lines as those above (that is, the same line as the minor device number minus 128).

A dial-in line has a minor device in the range 0 – 11 and is conventionally renamed **/dev/ttyd***n*, where *n* is a number indicating which dial-in line it is (so that **/dev/ttyd0** is the first dial-in line), and the dial-out line corresponding to that dial-in line has a minor device number 128 greater than the minor device number of the dial-in line and is conventionally named **/dev/cua***n*, where *n* is the number of the dial-in line.

The **/dev/cua***n* lines are special in that they can be opened even when there is no carrier on the line. Once a **/dev/cua***n* line is opened, the corresponding tty line can not be opened until the **/dev/cua***n* line is closed; a blocking open will wait until the **/dev/cua***n* line is closed (which will drop Data Terminal Ready, after which Carrier Detect will usually drop as well) and carrier is detected again, and a non-blocking open will return an error. Also, if the **/dev/ttyd***n* line has been opened successfully (usually only when carrier is recognized on the modem) the corresponding **/dev/cua***n* line can not be opened. This allows a modem to be attached to e.g. **/dev/ttyd0** (renamed from **/dev/ttya**) and used for dial-in (by enabling the line for login in **/etc/ttytab**) and also used for dial-out (by **tip**(1C) or **uucp**(1C)) as **/dev/cua0** when no one is logged in on the line. Note: the bit in the **flags** word in the configuration file (see above) must be zero for this line, which enables hardware carrier detection.

IOCTLS

The standard set of **termio ioctl**() calls are supported by **zs**.

If the CRTSCTS flag in the **c_cflag** is set, output will be generated only if CTS is high; if CTS is low, output will be frozen. If the CRTSCTS flag is clear, the state of CTS has no effect. Breaks can be generated by the TCSBRK, TIOCSBRK, and TIOCCBRK **ioctl**() calls. The modem control lines TIOCM_CAR, TIOCM_CTS, TIOCM_RTS, and TIOCM_DTR are provided.

The input and output line speeds may be set to any of the speeds supported by **termio**. The speeds cannot be set independently; when the output speed is set, the input speed is set to the same speed.

ERRORS

An **open**() will fail if:

ENXIO	The unit being opened does not exist.
EBUSY	The dial-out device is being opened and the dial-in device is already open, or the dial-in device is being opened with a no-delay open and the dial-out device is already open.
EBUSY	The unit has been marked as exclusive-use by another process with a TIOCEXCL **ioctl**() call.
EINTR	The open was interrupted by the delivery of a signal.

FILES

/dev/tty{a,b,s[0-3]}	hardwired tty lines
/dev/ttyd[0-9a-f]	dial-in tty lines
/dev/cua[0-9a-f]	dial-out tty lines

SEE ALSO

tip(1C), **uucp**(1C), **mcp**(4S), **mti**(4S), **termio**(4), **ldterm**(4M), **ttcompat**(4M), **ttysoftcar**(8)

DIAGNOSTICS

zs*n c* **: silo overflow.**

The 8530 character input silo overflowed before it could be serviced.

zs*n c* **: ring buffer overflow.**

The driver's character input ring buffer overflowed before it could be serviced.

Notes