NAME

console, keyboard, log − system console

DESCRIPTION

The TTY device driver manages two devices related to the main user interface, the system screen and the keyboard. These two together are named "the Console".

The Screen

The screen of a PC can be managed by a Monochrome Display Adapter, a Hercules card, a Color Graphics Adapter, an Enhanced Graphics Adapter, or a Video Graphics Array. To the console driver these devices are seen as a block of video memory into which characters can be written to be displayed, an I/O register that sets the video memory origin to the character that is to be displayed on the top-left position of the screen, and an I/O register that sets the position of the hardware cursor. Each character within video memory is a two-byte word. The low byte is the character code, and the high byte is the "attribute byte", a set of bits that controls the way the character is displayed, character and background colours for a colour card, or intensity/underline/reverse video for monochrome.

These are the characteristics of the adapters in text mode:

Adapter	Usable memory	Mono/Colour
MDA	4K	M
Hercules	4K	M
CGA	16K	C
EGA	32K	M or C
VGA	32K	M or C

MDA and Hercules are the same to the console driver, because the graphics mode of the Hercules is of no use to Minix. EGA and VGA are also mostly seen as the same in text mode. An EGA adapter is either a monochrome or a colour device depending on the screen attached to it. A VGA adapter can run in either monochrome or colour (grayscale) mode depending on how the Boot Monitor has initialized it.

The driver uses the video origin to avoid copying the screen contents when scrolling up or down. Instead, the origin is simply moved one line. This is named "hardware scrolling", as opposed to copying memory: "software scrolling".

The video origin is also used to implement several virtual consoles inside the video memory of the adapter. Each virtual console gets a segment of video memory. The driver chooses which console to display by moving the video origin. Note that an MDA or Hercules adapter can only support one console. CGA can support up to four 80x25 consoles, and EGA and VGA can have eight. It is best to configure one less console to leave some video memory free so that hardware scrolling has some space to work in.

Character codes are used as indices into a display font that is stored in the adapter. The default font is the IBM character set, which is an ASCII character set in the low 128 codes, and a number of mathematical, greek, silly graphics, and accented characters in the upper 128 codes. This font is fixed in the MDA, Hercules and CGA adapters, but can be replaced by a user selected font for the EGA and VGA adapters.

A number of control characters and escape sequences are implemented by the driver. The result is upward compatible with the ANSI standard terminal. The **termcap**(5) type is **minix**. Normal characters written to the console are displayed at the cursor position and the cursor is advanced one column to the right. If an entire line is filled then the cursor wraps to the first column of the next line when the next character must be displayed. The screen is scrolled up if needed to start a new line. Some characters have special effects when sent to the console. Some even have arguments in the form of comma separated decimal numbers. These numbers default to the lowest possible value when omitted. The top-left character is at position (1, 1). The following control characters and escape sequences are implemented by the console:

Sequence	Name	Function
ˆ@	Null	Ignored (padding character)

^G	Bell	Produce a short tone from the speaker
^H	Backspace	Move the cursor back one column, wrapping from the left edge up one line to the right edge
^I	Horizontal Tab	Move to the next tab stop, with each tab stop at columns 1, 9, 25, etc. Wrap to the next line if necessary.
^J	Line Feed	Move one line down, scrolling the screen up if necessary
^K	Vertical Tab	Same as LF
^L	Form Feed	Same as LF
^M	Carriage Return	Move to column 1
^[	Escape	Start of an escape sequence
^[M	Reverse Index	Move one line up, scrolling the screen down if necessary
^[[*n*A	Cursor Up	Move the cursor up *n* lines
^[[*n*B	Cursor Down	Move the cursor down *n* lines
^[[*n*C	Cursor Forward	Move the cursor right *n* columns
^[[*n*D	Cursor Backward	Move the cursor left *n* columns
^[[*m*;*n*H	Cursor Position	Move the cursor to line *m*, column *n*
^[[*s*J	Erase in Display	Clear characters as follows: *s* = 0: From cursor to end of screen *s* = 1: From start of screen to cursor *s* = 2: Entire screen
^[[*s*K	Erase in Line	Clear characters as follows: *s* = 0: From cursor to end of line *s* = 1: From start of line to cursor *s* = 2: Entire line
^[[*n*L	Insert Lines	Insert *n* blank lines
^[[*n*M	Delete Lines	Delete *n* lines
^[[*n*@	Insert Characters	Insert *n* blank characters
^[[*n*P	Delete Characters	Delete *n* characters
^[[*n*m	Character Attribute	Set character attribute as follows: *n* = 0: Normal (default) attribute *n* = 1: Bold (mono) / Yellow (colour) *n* = 4: Underline (M) / Light green (C) *n* = 5: Blinking (M) / Magenta (C) *n* = 7: Reverse Video *n* = 30: Black foreground colour *n* = 31: Red *n* = 32: Green *n* = 33: Orange *n* = 34: Blue *n* = 35: Magenta *n* = 36: Light blue *n* = 37: White *n* = 40 − 47: Same for background colour

The console device implements the following ioctl to copy a font into font memory on EGA and VGA adapters:

ioctl(*fd***, TIOCSFON, u8_t** *font***[256][32]);**

Font memory consists of 256 character definitions of 32 lines per character and 8 pixels per line. The first line is the topmost line of the character. The leftmost pixel is lit if the most significant bit of a line is set, etc. The 80x25 video mode used by Minix has an 8x16 character cell, which means that only the first 16 lines of a character are displayed.

The Keyboard

The keyboard produces key codes for each key that is pressed. These keys are transformed into character codes or sequences according to the current keyboard translation table. The format of this table is described in **keymap**(5). The character codes can be read from the console device unless they map to special hotkeys. The hotkeys are as follows:

Name	KeyFunction
CTRL−ALT−DEL	Send an abort signal to process 1 (init). Init then halts the system
CTRL−ALT−KP-.	Likewise for keypad period
F1	Process table dump
F2	Show memory map
F3	Toggle software/hardware scrolling
F5	Show network statistics
CTRL−F7	Send a quit signal to all processes connected to the console
CTRL−F8	Send an interrupt signal
CTRL−F9	Send a kill signal. If CTRL−F8 or CTRL−F7 don't get 'em, then this surely will. These keys are for disaster recovery. You would normally use DEL and CTRL−\ to send interrupt and quit signals.
ALT−F1	Select virtual console 0 (/dev/console)
ALT−F2	Select virtual console 1 (/dev/ttyc1)
ALT−F(*n*+1)	Select virtual console *n* (/dev/ttyc*n*)
ALT−Left	Select previous virtual console
ALT−Right	Select next virtual console

The keyboard map is set with the **KIOCSMAP** ioctl whose precise details are currently hidden in the **loadkeys** utility.

Log device

The **log** device can be used by processes to print debug messages onto the console. The console is a terminal type device, so it is taken from processes when a session leader exits. This does not happen with the log device.

SEE ALSO

tty(4), **loadkeys**(1), **keymap**(5), **boot**(8).

NOTES

Output processing turns Line Feeds into CR LF sequences. Don't let this surprise you. Either turn off output processing or use one of the synonyms for LF.

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

dev − device files in /dev

DESCRIPTION

Device files are the eyes and ears of the system. Through the device files one has access to the disks, terminals and other parts of the machine. Single bytes or disk blocks may be transferred to or from a device with ordinary **read**(2) or **write**(2) calls, byte positions set with **lseek**(2), or more complicated control functions performed with **ioctl(2).**

Device files as found in **/dev** have several attributes that must be considered. Here are two examples as **ls −l** shows them:

```
brw-rw-rw-  1 root     operator   2,   1 Jun 10  1995 fd1
crw--w----  1 kjb      tty        4,   0 May 11 09:41 console
```

Most attributes are the same as for a regular file and have the same function. The file type and the major and minor device numbers are special to devices.

Character devices are marked with a **c** as a file type letter. Any I/O on a character device is sent down to the device driver without any interpretation. This means that a process doing the I/O must know the characteristics of the device and deal with them appropriately.

Block devices provoke the file system server into buffering the data on those devices. Data read or written by processes is passed through the file system block cache. Unaligned bytes read or written are extracted or reassembled by the file server from or to whole blocks in the cache. The file server transfers data to or from the device driver as blocks to positions at block size boundaries. These blocks are Minix blocks of 1024 bytes, disk devices usually have a 512 byte block size. Only block devices can be mounted as part of the file system tree if they contain a Minix file system.

The major device number (2 for **fd1** and 4 for **console**) are used by FS to find the device driver that manages a device. The minor device number (1 for **fd1** and 0 for **console**) is passed to the driver to select a device among a number of related devices that are all managed by that driver. The device drivers are usually kernel tasks under Minix, small processes that are contained within the address space of the kernel. The following tasks and associated devices exist:

Memory (major 1)

The **ram**, **mem**, **kmem**, and **null** devices are managed by the memory task. The **ram** device is a block device for a chunk of memory that is the RAM disk. Any byte read from or written to the **ram** device is copied from or to that memory chunk. The **mem** device is a character device for the entire address space of the system, but **kmem** only for the kernel data area. These two devices allow programs like **ps**(1) to hunt around the system looking for interesting bits. The **null** device is a data sink. It happily swallows any bytes written to it, and returns nothing on a read.

Floppy disk (major 2)

The **fd0**, **fd0a**, **fd0b**, **fd0c**, and **fd0d** block devices are the first floppy disk and the four partitions that may exist on a that floppy disk. Likewise are **fd1** and **fd1[a−d]** the device and partitions for the second floppy disk. The floppy disk devices are described in detail in **fd**(4). Partitioning in general is explained in **hd**(4).

Hard disk (major 3)

The first hard disk can be accessed by block device **hd0**. This device addresses the entire hard disk from the first to the last sector. A hard disk is normally partitioned in up to four primary partitions, **hd1**, **hd2**, **hd3**, and **hd4**. Each of these devices accesses a range of sectors on the **hd0** device. It is customary to give each operating system on a disk a primary partition. So the MS-DOS C: "drive" can be on **hd1**, and Minix can be on **hd2**. Minix wants to have several partitions on its own, so **hd2** can be further subdivided into the subpartitions **hd2a**, **hd2b**, **hd2c**, and **hd2d**. **/dev** contains devices for the first and second hard disk (**hd0** and **hd5**) their primary partitions (**hd[1−46−9]**) and subpartitions thereof (**hd[1−46−9][a−d]**). More detail can be found in **hd**(4).

Terminals (minor 4)

The TTY driver manages the system console device, aptly named **console**, the serial lines, **tty00** and **tty01**, and the pseudo ttys. Through the console device one can display characters on a screen attached to a monochrome, Hercules, color, or VGA adapter. The **ttyc1**, **ttyc2**, etc. devices are the so-called "virtual consoles" that share the one console display. One can select which virtual console is to be visible on the screen and take input from the keyboard. To allow remote login the devices with minor numbers of 128 or higher offer virtual terminals. These pseudo ttys come in tty, pty pairs that form a pipe between processes running under the tty, and a controlling process attached to the pty side. See also **console**(4), and **tty**(4).

Anonymous TTY (major 5)

This is just one device named **tty** that is a synonym for the controlling tty of a process. This device is not managed by any device driver, but is handled by FS itself. A process can get access to the terminal it is running under by using **/dev/tty**.

Line printer (major 6)

The **lp** device sends any bytes written to it to the printer.

TCP/IP (major 7)

The TCP/IP task is not a kernel task, but a server like MM and FS. It sits between FS and the DP8390 task that manages the ethernet boards. Together they implement the TCP/IP protocol. See also **ip**(4).

CD-ROM (major 8)

This is the CD-ROM driver for the Mitsumi proprietary CD-ROM interface. The **cd0** device addresses the whole CD, with extra **cd[1−4]** and **cd[1−4][a−d]** devices for if the CD also contains partitions with Minix file systems.

SCSI disks and tapes (major 10)

The **sd*** devices are disks in the same way as the **hd*** devices. Except that these disks are SCSI disks attached to an Adaptec 1540 controller or compatible. The driver also manages the **rst*** and **nrst*** tape devices (rewinding or non-rewinding). See **sd**(4).

Audio (major 13)

The **audio** device can be used to produce or record air vibrations using a Soundblaster 16 type audio card. See **audio**(4).

Mixer (major 14)

The **mixer** device is used to control the audio driver.

FILES

/dev/* All Minix devices

SEE ALSO

read(2), **write**(2), **lseek**(2), **ioctl**(2), **console**(4), **fd**(4), **hd**(4), **ip**(4), **sd**(4), **tty**(4), **MAKEDEV**(8).

DIAGNOSTICS

There are five prominent errors that processes accessing device files may provoke:

ENODEV − No such device

There is no driver managing the device class this device belongs to. Either the driver is configured out, or it is not loaded (inet).

ENXIO − No such device or address

This device is not available. Either the driver does not support it at all, or the hardware isn't available, i.e. accessing the second disk on a system with only one disk.

EACCES − Permission denied

This error may cause a lot of head scratching if **ls −l** shows a device file to be writable. The media you are trying to access is simply physically write protected!

EINVAL − Invalid argument

Devices may not like reads or writes that are not block multiples, or very big transfers, etc. The device manual page should list the limits.

EIO − I/O error

This may be a real I/O error, i.e. a read or write on the device failing due to a media error. But it may also be the result of an operation that a device can't do, or an empty tape drive, etc.

NOTES

Some devices are not present by default. The **MAKEDEV** script knows how to make them.

MS-DOS equivalents

The names of MS-DOS devices map to Minix devices as follows:

A:	fd0	
B:	fd1	
C:	hd1, sd1	(usually the first partition)
D:	hd6, sd1, sd6	(or an extended partition)
CON	console	
COM1	tty00	(UNIX counts from 0)
LPT1	lp	

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

fd − floppy disk

DESCRIPTION

The **fd*** devices refer to the Floppy disk driver using the NEC PD765 floppy disk controller. These diskettes are arrays of 512 byte sectors, although Minix always works with two sectors at a time due to its 1024 byte block size. You can read or write any number of bytes however, Minix takes care of cutting and pasting incomplete blocks together.

The driver is normally configured for two floppy disk devices **fd0** and **fd1**. It can handle two more, but it is unlikely that the average PC can.

On the first access to an **fd** device (by **open**(2) or **mount**(2)), the driver will execute a series of read tests to determine the floppy type. This works ok for all floppy types except the true 360k type, because it is indistinguishable from the 720k type. This only means that the size of the floppy is not estimated right.

Bits 2−6 of the minor device number may be set to the floppy disk type to make it known to the driver what type of diskette it is reading or writing. The non-auto devices should be used for formatting, or when one wants to be absolutely sure that the device is accessed right. These devices exist for drive 0:

type	device	minor	media
0	fd0	0	autodetect
1	pc0	4	360k, 5.25"
2	at0	8	1.2M, 5.25"
3	qd0	12	360k in a 720k, 5.25" drive
4	ps0	16	720k, 3.5"
5	pat0	20	360k in a 1.2M, 5.25" drive
6	qh0	24	720k in a 1.2M, 5.25" drive
7	PS0	28	1.44M, 3.5"

Type 4 may also be used for the rarely seen 720k, 5.25" floppies (type 2 made them obsolete fast.) Note that these "types" only describe the floppies from a software point of view, type 1 and 4 drives use the same parameters.

If the format bit (bit 7) is set, then the driver interprets write commands as track formatting requests. This is used by the **format**(1) command.

If the type bits are set to 28, 29, 30, or 31, then the driver uses a partition table found in sector 0 to partition the floppy. The partitions of **fd0** may be accessed as **fd0a** through **fd0d**. See **hd**(4) for a description of the partition table, and associated ioctl commands.

FILES

/dev/fd[0−3], /dev/pc[0−3], /dev/at[0−3], /dev/qd[0−3], /dev/ps[0−3], /dev/pat[0−3], /dev/qh[0−3], /dev/PS[0−3], /dev/fd[0−3][a−d]

SEE ALSO

format(1), **hd**(4), **part**(8).

BUGS

The driver does not know the size of a 360k diskette in a 360k 5.25" drive, because it uses the 720k parameters for it. So it will happily try to read past the end making all kinds of interesting noises. It’s a good thing these drives are practically obsolete.

The partition table is only read when the drive motor is off and only for an auto or partition device. The driver assumes that a floppy in a drive with a running motor can’t have been replaced all of a sudden.

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

hd − winchester hard disk

DESCRIPTION

The **hd*** family of devices refer to the Winchester hard disk drivers for the IBM XT, AT and PS/2 machines, but may also refer to the generic (and slower) BIOS based hard disk driver. These disks are arrays of 512 byte sectors, although Minix always works with two sectors at a time due to its 1024 byte block size. You can read or write any number of bytes however, Minix takes care of cutting and pasting incomplete blocks together.

The devices may be divided into three classes:

The devices with a minor device number that is a multiple of 5, i.e. **hd0** or **hd5**, refer to the whole hard disk 0 and 1. Through these devices one has access to any block on the hard disk. Most notably the partition table, that can be found in the first sector of the disk.

The devices with a minor device number that is not a multiple of 5, i.e. **hd1**, **hd2**, ..., **hd6**, ..., refer to primary partitions of the lower numbered whole hard disk device. These devices normally contain MS-DOS or Minix file systems. /dev/hd1 is often the MS-DOS C: drive.

Minor devices from 128 up may refer to Minix subpartitions within primary partitions if a subpartition table has been placed in a Minix primary partition. The subpartitions of **hd3** for instance, are named **hd3a** through **hd3d**. Their minor device numbers may be calculated as 128 + 16*drive + 4*partition + subpartition, counting the partitions from zero.

If a primary partition is an extended partition then up to four logical partitions can be accessed as subpartitions of that extended partition. This allows one to access foreign file systems of other operating systems, Minix file systems are not normally placed in logical partitions.

PARTITIONING

The first sector of a drive (or partition for subpartitioning) contains the partition table at byte offset 446. This is what each of the four entries looks like as defined in <ibm/partition.h>:

```
/* Description of entry in the partition table.  */

struct part_entry {
   unsigned char bootind;          /* boot indicator 0/ACTIVE_FLAG         */
   unsigned char start_head;       /* head value for first sector          */
   unsigned char start_sec;        /* sector value + high 2 cyl bits       */
   unsigned char start_cyl;        /* low 8 cylinder bits                  */
   unsigned char sysind;           /* system indicator                     */
   unsigned char last_head;        /* h/s/c for the last sector            */
   unsigned char last_sec;
   unsigned char last_cyl;
   unsigned long lowsec;           /* logical first sector                 */
   unsigned long size;             /* size of partition in sectors         */
};

#define ACTIVE_FLAG      0x80      /* value for active in bootind field    */
#define NR_PARTITIONS    4         /* number of entries in table           */
#define PART_TABLE_OFF 0x1BE       /* offset of table in boot sector       */

/* Partition types (sysind). */
#define MINIX_PART       0x81      /* Minix partition type */
#define NO_PART          0x00      /* unused entry */
#define OLD_MINIX_PART 0x80        /* created before 1.4b, obsolete */
#define EXT_PART         0x05      /* extended partition */
```

The cylinder numbers are encoded in a very strange way, bits 8 and 9 are in the high two bits of the sector number. The sector numbers count from 1, not 0! More useful are the lowsec and size fields however, they simply give the location of the partition as an absolute sector offset and length within the drive.

The partition table entry defined above is specific to IBM type disks. The device drivers use another partition entry structure to pass information on a partition. This is what <minix/partition.h> looks like:

```
struct partition {
   u64_t base;              /* byte offset to the partition start */
   u64_t size;              /* number of bytes in the partition */
   unsigned cylinders;      /* disk geometry for partitioning */
   unsigned heads;
   unsigned sectors;
};
```

The base and size fields are the byte offset and length of a partition. (These are 64 bit numbers under Minix-vmd, but only 32 bit numbers under standard Minix.) The geometry of the disk is also given for the benefit of partition table editors. This information can be obtained from an open disk device with the call:

ioctl(*fd***, DIOCGETP, &***entry***);**

One can change the placement of the device to the lowsec and size fields of *entry* by using the **DIOCSETP** call instead. Only the base and size fields are used for **DIOCSETP**.

The partition tables when read from disk by the driver are checked and truncated to fit within the primary partition or drive. The first sector should be left free for the partition table.

The partition tables are read when the in-use count (opens and mounts) changes from 0 to 1. So an idle disk is automatically repartitioned on the next access. This means that repartitioning programs only have effect if a disk stays in use, unless they reload a changed partition table.

FILES

/dev/hd[0−9], /dev/hd[1−46−9][a−d]

SEE ALSO

ioctl(2), **int64**(3), **part**(8), **repartition**(8).

BUGS

The subpartitioning is incompatible with the MS-DOS method of extended partitions. The latter does not map well to the sparse minor device number space.

The primary partition table is sorted by lowsec like MS-DOS does, subpartition tables are not. Just think about what happens when you delete a partition in the MS-DOS scheme.

Don't move a partition that is mounted or kept open by some process. The file system may write cached blocks to the new location.

The BIOS driver is not slow at all on a buffered disk.

Some IDE disks send an interrupt when they spin down under hardware power management. The driver acknowledges the interrupt as it is supposed to do by reading the status register. The disk then spins up again... You have to disable the spin down in the computer setup to fix the problem.

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

ip, eth, psip, udp, tcp − Internet Protocol server devices and definitions

DESCRIPTION

The **ip***, **eth***, **psip***, **tcp***, and **udp*** devices give access to the Internet Protocol (IP) services in Minix. There are 16 such devices for four different networks, and four different ways of accessing them. The **eth*** and **psip*** devices give direct access to the network packets at the lowest level. The **ip***, **tcp***, and **udp*** devices give access to IP, TCP, or UDP services. See **set_net_default**(8) for a description of the default network devices and environment variables. The proper TCP device to use for instance is found by:

```
if ((tcp_device= getenv("TCP_DEVICE")) == NULL)
        tcp_device= "/dev/tcp";
```

Access to the IP services is provided using filedescriptors to open IP devices. These open IP channels can be configured with **ioctl**(2) calls, and data can be transferred by calls to **read**(2), and **write**(2).

Types (general)

<sys/types.h>
defines **u8_t**, **u16_t**, **u32_t** and **i32_t** (and **U8_t**, **U16_t**, **U32_t** and **I32_t** for use in prototypes).

Types (eth)

<net/gen/ether.h>
defines struct ether_addr (**ether_addr_t**) and **ether_type_t** and **Ether_type_t** for use in prototypes.

<net/gen/eth_io.h>
defines struct nwio_ethopt (**nwio_ethopt_t**) and struct nwio_ethstat (**nwio_ethstat_t**)

<net/gen/eth_hdr.h>
defines struct eth_hdr (**eth_hdr_t**)

Types (ip)

<net/gen/in.h>
defines **ipaddr_t**, **ipproto_t** and struct ip_hdropt (**ip_hdropt_t**).

<net/gen/ip_io.h>
defines struct nwio_ipconf (**nwio_ipconf_t**) and struct nwio_ipopt (**nwio_ipopt_t**)

<net/gen/ip_hdr.h>
defines struct ip_hdr (**ip_hdr_t**)

<net/gen/route.h>
defines struct nwio_route (**nwio_route_t**)

Types (tcp)

<net/gen/tcp.h>
defines **tcpport_t** and **Tcpport_t** for use in prototypes.

<net/gen/tcp_io.h>
defines struct nwio_tcpconf (**nwio_tcpconf_t**), struct nwio_tcpcl (**nwio_tcpcl_t**), struct nwio_tcpatt (**nwio_tcpatt_t**) and struct nwio_tcpopt (**nwio_tcpopt_t**).

<net/gen/tcp_hdr.h>
defines struct tcp_hdr (**tcp_hdr_t**) and struct tcp_hdropt (**tcp_hdropt_t**).

Types (udp)

<net/gen/udp.h>
defines **udpport_t** and **Udpport_t** for use in prototypes.

<net/gen/udp_io.h>
defines struct nwio_udpopt (**nwio_udpopt_t**).

<net/gen/udp_hdr.h>
defines struct udp_hdr (**udp_hdr_t**) and struct udp_io_hdr (**udp_io_hdr_t**).

Byte Order Conversion

All 16-bit and 32-bit quantities in IP headers must be in network byte order. The macros described in **hton**(3) can be used to convert these values to and from the byte order used by the host machine.

The Internet Checksum

The **OneC_sum** function (see **oneC_sum**(3)) is used to calculate the one's complement checksum needed for IP network packets.

General Functions

fd **= open(***tcpip_device***, O_RDWR)**

This is how one normally obtains a filedescriptor for a new TCP/IP channel. *tcpip_device* names one of the TCP/IP devices. The channel may be used both to send or to receive data.

n **= read(***fd***,** *buf***,** *size***)**

Receives one packet (low level devices) or a number of bytes (TCP stream). Returns the the number of bytes placed into *buf*, or returns -1 with an error code placed into **errno**.

n **= write(***fd***,** *buf***,** *size***)**

Sends one packet (low level devices) or a number of bytes (TCP stream). Returns *size* or -1 with the error code placed into **errno**. The TCP/IP **read** and **write** functions behave like reads and writes on pipes when it comes to signals.

ETH Functions

ioctl(*fd***, NWIOGETHSTAT, &struct nwio_ethstat)**

The **NWIOGETHSTAT** ioctl returns the Ethernet address and some statistics of the Ethernet server of the channel *fd*. The result is returned in the nwio_ethstat structure. The **struct nwio_ethstat** is defined in <net/gen/eth_io.h>:

```
typedef struct nwio_ethstat
{
        ether_addr_t nwes_addr;
        eth_stat_t nwes_stat;
} nwio_ethstat_t;

typedef struct eth_stat
{
    unsigned long ets_recvErr,  /* # receive errors */
      ets_sendErr,            /* # send error */
      ets_OVW,                /* # buffer overwrite warnings,
                                   (packets arrive faster than
                                    can be processed) */
      ets_CRCerr,             /* # crc errors of read */
      ets_frameAll,           /* # frames not aligned (# bits
                                   not a multiple of 8) */
      ets_missedP,            /* # packets missed due to too
                                   slow packet processing */
      ets_packetR,            /* # packets received */
      ets_packetT,            /* # packets transmitted */
      ets_transDef,           /* # transmission deferred (there
                                   was a transmission of an
                                   other station in progress */
      ets_collision,          /* # collisions */
      ets_transAb,            /* # transmissions aborted due
                                   to excessive collisions */
      ets_carrSense,          /* # carrier sense lost */
      ets_fifoUnder,         /* # fifo underruns (processor
```

```
                                        is too busy) */
        ets_fifoOver,           /* # fifo overruns (processor is
                                        too busy) */
        ets_CDheartbeat,         /* # times unable to transmit
                                        collision signal */
        ets_OWC;                 /* # times out of window
                                        collision */
} eth_stat_t;
```

ioctl(*fd***, NWIOSETHOPT, &struct nwio_ethopt)**

Before an Ethernet channel can be used to send or receive Ethernet packets, it has to be configured using the **NWIOSETHOPT** ioctl. The structure **nwio_ethopt** is defined in <net/gen/eth_io.h>:

```
typedef struct nwio_ethopt
{
        u32_t nweo_flags;
        ether_addr_t nweo_multi, nweo_rem;
        ether_type_t nweo_type;
} nwio_ethopt_t;

#define NWEO_NOFLAGS    0x0000L
#define NWEO_ACC_MASK   0x0003L
#       define NWEO_EXCL        0x00000001L
#       define NWEO_SHARED      0x00000002L
#       define NWEO_COPY        0x00000003L
#define NWEO_LOC_MASK   0x0010L
#       define NWEO_EN_LOC      0x00000010L
#       define NWEO_DI_LOC      0x00100000L
#define NWEO_BROAD_MASK 0x0020L
#       define NWEO_EN_BROAD    0x00000020L
#       define NWEO_DI_BROAD    0x00200000L
#define NWEO_MULTI_MASK 0x0040L
#       define NWEO_EN_MULTI    0x00000040L
#       define NWEO_DI_MULTI    0x00400000L
#define NWEO_PROMISC_MASK 0x0080L
#       define NWEO_EN_PROMISC  0x00000080L
#       define NWEO_DI_PROMISC  0x00800000L
#define NWEO_REM_MASK   0x0100L
#       define NWEO_REMSPEC     0x00000100L
#       define NWEO_REMANY      0x01000000L
#define NWEO_TYPE_MASK  0x0200L
#       define NWEO_TYPESPEC    0x00000200L
#       define NWEO_TYPEANY     0x02000000L
#define NWEO_RW_MASK    0x1000L
#       define NWEO_RWDATONLY   0x00001000L
#       define NWEO_RWDATALL    0x10000000L
```

The configuration is divided in a number of section (covered by the xx_MASK macros). Options can be set in the **nweo_flags** field. The first section (**NWEO_ACC_MASK**) controls the access to a certain Ethernet packet type. If **NWEO_EXCL** is selected then this is the only channel that can send or receive Ethernet packets of the selected type. If **NWEO_SHARED** is selected then multiple channels (which all have to select **NWEO_SHARED**) can use the same Ethernet type, they all can send packets but incoming packets will be delivered to at most one of them. If **NWEO_COPY** is selected then multiple channels have access to the same Ethernet type and all receive a copy of an incoming packet.

The **NWEO_LOC_MASK** flags control the delivery of packets with a destination address equal to the Ethernet address of the machine. If **NWEO_EN_LOC** is selected then these packets will be delivered and with **NWEO_DI_LOC** they will be discarded.

NWEO_BROAD_MASK, **NWEO_MULTI_MASK**, and **NWEO_PROMISC_MASK** do the same to broadcast packets, multicast packets and promiscuous mode packets as **NWEO_LOC_MASK** does for local packets. Except that the precise multicast address is taken from the **nweo_multi** field.

The **NWEO_REM_MASK** flags control whether communication is restricted to single destination or not. **NWEO_REMSPEC** restricts sending and receiving of packets to the single remote computer specified in the **nweo_rem** field. **NWEO_REMANY** allows sending to and receiving from any remote computer.

NWEO_TYPESPEC restricts sending and receiving of packets to the type specified in **nweo_type**. The type has to be in network byte order (using **hton**(3)). **NWEO_TYPEANY** allows any type.

If the Ethernet header is completely specified by the **nweo_flags** i.e., all of **NWEO_EN_LOC**, **NWEO_DI_BROAD**, **NWEO_DI_MULTI**, **NWEO_DI_PROMISC**, **NWEO_REMSPEC** and **NWEO_TYPESPEC** are specified, then **NWEO_RWDATONLY** can be used to send and receive only the data part of an Ethernet packet. If **NWEO_RWDATALL** is specified then both Ethernet header and data are used.

PSIP Functions

[[[No description available yet.]]]

IP Functions

ioctl(*fd***, NWIOGIPCONF, &struct nwio_ipconf)**

The **NWIOGIPCONF** ioctl reports the Internet Address and the netmask. For the *nwio_ipconf* structure see the **NWIOSIPCONF** ioctl below.

ioctl(*fd***, NWIOGIPOROUTE, &struct nwio_route)**

The **NWIOGIPOROUTE** ioctl can be used to query an IP server about its routing table. [[[NWIODIPOROUTE, NWIOGIPIROUTE, NWIODIPIROUTE?]]] The structure **nwio_route** is defined in <net/gen/route.h>:

```
typedef struct nwio_route
{
        u32_t nwr_ent_no;
        u32_t nwr_ent_count;
        ipaddr_t nwr_dest;
        ipaddr_t nwr_netmask;
        ipaddr_t nwr_gateway;
        u32_t nwr_dist;
        u32_t nwr_flags;
        u32_t nwr_pref;
} nwio_route_t;

#define NWRF_EMPTY      0
#define NWRF_INUSE      1
#define NWRF_FIXED      2
```

The requested entry is taken from **nwr_ent_no**. Entries are counted from 0, so the value 0 can be used for an initial query. The size of the routing table is returned in **nwr_ent_count**. The **nwr_flags** indicates if the entry is in use (**NWRF_INUSE**) and if the entry was inserted manually (using **NWIOSIPOROUTE**) or generated by the IP server itself. The route is described by **nwr_dest**, **nwr_netmask**, **nwr_gateway**, **nwr_dist**, and **nwr_pref**. **Nwr_dest** and **nwr_netmask** select the destination addresses. A value of 0.0.0.0 (0x0) in both **nwr_dest** and **nwr_netmask** means every host. A value of 255.255.255.255 (0xffffffff) in **nwr_netmask** means a single host. Other values of **nwr_netmask** are netmasks for the network specified by **nwr_dest**. **Nwr_gateway** is gateway that should be used. **Nwr_dist** is a minimal distance. Packets with a time to live smaller than **nwr_dist** will not reach the destination. If two routes have equal netmask and distance fields but different gateways then the gateway with highest value in **nwr_pref** is used.

ioctl(*fd*, NWIOSIPCONF, &struct nwio_ipconf)

The **NWIOSIPCONF** ioctl can be used to inform the IP server about its Internet Address and/or its netmask. Normally an IP server will discover its Internet Address using the RARP protocol. **NWIOSIPCONF** can be used in the case that the RARP failed, or the netmask has to be changed. Note that higher level protocols (TCP and UDP) assume that the Internet Address of an IP device does not change, therefore TCP and UDP stop functioning if the Internet Address is changed.

The structure **nwio_ipconf** is defined in <net/gen/ip_io.h>:

```
typedef struct nwio_ipconf
{
      u32_t   nwic_flags;
      ipaddr_t nwic_ipaddr;
      ipaddr_t nwic_netmask;
} nwio_ipconf_t;

#define NWIC_NOFLAGS            0x0
#define NWIC_FLAGS              0x3
#       define NWIC_IPADDR_SET          0x1
#       define NWIC_NETMASK_SET         0x2
```

The function of **nwio_ipconf** depends on the value of **nwic_flags**. If **NWIC_IPADDR_SET** is set then the Internet Address will be set to **nwic_ipaddr**. If **NWIC_NETMASK_SET** is set then the Internet Address will be set to **nwic_netmask**.

ioctl(*fd*, NWIOSIPOPT, &struct nwio_ipopt)

Before an IP channel can be used, it has to be configured using the **NWIOSIPOPT** ioctl. The structure **nwio_ipopt** is defined in <net/gen/ip_io.h>:

```
typedef struct nwio_ipopt
{
        u32_t nwio_flags;
        ipaddr_t nwio_rem;
        ip_hdropt_t nwio_hdropt;
        u8_t nwio_tos;
        u8_t nwio_ttl;
        u8_t nwio_df;
        ipproto_t nwio_proto;
} nwio_ipopt_t;

#define NWIO_NOFLAGS    0x0000L
#define NWIO_ACC_MASK   0x0003L
#       define NWIO_EXCL        0x00000001L
#       define NWIO_SHARED      0x00000002L
#       define NWIO_COPY        0x00000003L
#define NWIO_LOC_MASK   0x0010L
#       define NWIO_EN_LOC      0x00000010L
#       define NWIO_DI_LOC      0x00100000L
#define NWIO_BROAD_MASK 0x0020L
#       define NWIO_EN_BROAD    0x00000020L
#       define NWIO_DI_BROAD    0x00200000L
#define NWIO_REM_MASK   0x0100L
#       define NWIO_REMSPEC     0x00000100L
#       define NWIO_REMANY      0x01000000L
#define NWIO_PROTO_MASK 0x0200L
#       define NWIO_PROTOSPEC   0x00000200L
#       define NWIO_PROTOANY    0x02000000L
#define NWIO_HDR_O_MASK 0x0400L
#       define NWIO_HDR_O_SPEC  0x00000400L
```

```
#       define NWIO_HDR_O_ANY   0x04000000L
#define NWIO_RW_MASK    0x1000L
#       define NWIO_RWDATONLY   0x00001000L
#       define NWIO_RWDATALL    0x10000000L
```

The options are divided in several categories: **NWIO_ACC_MASK**, **NWIO_LOC_MASK**, **NWIO_BROAD_MASK**, **NWIO_REM_MASK**, **NWIO_PROTO_MASK**, **NWIO_HDR_O_MASK** and **NWIO_RW_MASK**. A channel is configured when one option of each category is set.

The options covered by **NWIO_ACC_MASK** control the number of channels that can use one IP protocol. If **NWIO_EXCL** is specified then only that channel can use a certain IP protocol. If **NWIO_SHARED** then multiple channels that all have to specify **NWIO_SHARED** can use the same IP protocol, but incoming packets will be delivered to a most one channel. **NWIO_SHARED** does not impose any restrictions. Every channel gets a copy of an incoming packet.

NWIO_LOC_MASK and **NWIO_BROAD_MASK** control the delivery of packets. If **NWIO_EN_LOC** is specified then packets that are explicitly send to the IP server are delivered. If **NWIO_EN_BROAD** is specified then broadcast packets are delivered. Either one or both of them can be disabled with **NWIO_DI_LOC** and **NWIO_DI_BROAD**.

NWIO_REMSPEC can be used to restrict communication to one remote host. This host is taken from the **nwio_rem** field. If any remote host is to be allowed then **NWIO_REMANY** can be used.

NWIO_PROTOSPEC restricts communication to one IP protocol, specified in **nwio_proto**. **NWIO_PROTOANY** allows any protocol to be sent or received.

NWIO_HDR_O_SPEC specifies all IP header options in advance. The values are taken from **nwio_hdropt**, **nwio_tos**, **nwio_ttl**, and **nwio_df**. **Nwio_hdropt** specifies the IP options that should be present in an outgoing packet. **Ip_hdropt_t** is defined in <net/gen/in.h>:

```
typedef struct ip_hdropt
{
        u8_t iho_opt_siz;
        u8_t iho_data[IP_MAX_HDR_SIZE-IP_MIN_HDR_SIZE];
} ip_hdropt_t;
```

The bytes of size **iho_opt_siz** in **iho_data** are appended to the IP header. **Nwio_tos** specifies the value of the ‘‘type of service’’ bits, **nwio_ttl** gives the value of the ‘‘time to live’’ field and **nwio_df** specifies whether fragmentation is disallowed or not. **NWIO_HDR_O_ANY** specifies that the header options should be specified at each write request.

NWIO_RWDATONLY specifies that the header should be omitted from a write request. This option can only be used when all header fields are specified in previous options: **NWIO_EN_LOC**, **NWIO_DI_BROAD**, **NWIO_REMSPEC**, **NWIO_PROTOSPEC** and **NWIO_HDR_O_SPEC**. A read operation will also only return the data part, so the IP options will be lost.

ioctl(*fd***, NWIOSIPOROUTE, &struct nwio_route)**

The **NWIOSIPOROUTE** ioctl adds a route to the routing table. See **NWIOGIPOROUTE** above for a description of the **nwio_route** structure. The fields **nwr_ent_no** and **nwr_ent_count** are ignored.

TCP Functions

ioctl(*fd***, NWIOTCPCONN, &struct nwio_tcpcl)**

The **NWIOTCPCONN** ioctl tries to setup a connection with a remote TCP/IP server. The channel must be fully configured (see **NWIOSTCPCONF**) and values for the local port, the remote port and the remote address have be specified using **NWTC_LP_SET** or **NWTC_LP_SEL**, **NWTC_SET_RA** and **NWTC_SET_RP**. The struct nwio_tcpcl is defined in <net/gen/tcp_io.h> as:

```
typedef struct nwio_tcpcl
{
      long nwtcl_flags;
      long nwtcl_ttl;
```

```
} nwio_tcpcl_t;
```

Set the **nwtcl_flags** field to zero before the connect or listen call. [[[Further explanation of nwio_tcpcl?]]]

ioctl(*fd*, NWIOGTCPCONF, &struct nwio_tcpconf)

This call reports the current configuration of a TCP channel. The **nwtc_flags** field shows the status of the **access**, **locport**, **remaddr** and **remport** fields. **Nwtc_locaddr** contains the Internet address of the TCP/IP server. **Remaddr** contains the Internet address of the remote TCP/IP server when set with **NWTC_SET_RA** or after a successful connect or listen (see **NWIOTCPCONN** or **NWIOTCPLISTEN**). **Nwio_locport** contains the local TCP/IP port set with **NWTC_LP_SET** or the selected port set with **NWTC_LP_SEL**. **Nwtc_remport** contains the TCP port of the remote TCP/IP server as set with **NWIO_SET_RP** or after a successful connect or listen.

A value of 0 (zero) is reported for **nwtc_remaddr**, **nwtc_locport** or **nwtc_remport** when no value is set either explicitly or implicitly.

ioctl(*fd*, NWIOTCPLISTEN, &struct nwio_tcpcl)

The **NWIOTCPLISTEN** ioctl waits until a remote TCP/IP server tries to connect to this channel. The channel has to be configured (see **NWIOSTCPCONF**). An additional restriction is that the local port must be set (with **NWTC_LP_SET**) or selected (with **NWTC_LP_SEL**). When a remote address is set only connections for that host are accepted, and when a remote port is set only connections from that port are accepted. After a successful listen **NWIOGTCPCONF** can be used to find out what the address and port of the other side are.

ioctl(*fd*, NWIOSTCPCONF, &struct nwio_tcpconf)

Before a TCP channel can be used it must configured using the **NWIOSTCPCONF** ioctl. The parameters to **NWIOSTCPCONF** are the channel file descriptor and a **struct nwio_tcpconf** as defined in <net/gen/tcp_io.h>:

```
typedef struct nwio_tcpconf
{
      u32_t nwtc_flags;
      ipaddr_t nwtc_locaddr;
      ipaddr_t nwtc_remaddr;
      tcpport_t nwtc_locport;
      tcpport_t nwtc_remport;
} nwio_tcpconf_t;

#define NWTC_NOFLAGS    0x0000L
#define NWTC_ACC_MASK   0x0003L
#       define NWTC_EXCL        0x00000001L
#       define NWTC_SHARED      0x00000002L
#       define NWTC_COPY        0x00000003L
#define NWTC_LOCPORT_MASK       0x0030L
#       define NWTC_LP_UNSET    0x00000010L
#       define NWTC_LP_SET      0x00000020L
#       define NWTC_LP_SEL      0x00000030L
#define NWTC_REMADDR_MASK       0x0100L
#       define NWTC_SET_RA      0x00000100L
#       define NWTC_UNSET_RA    0x01000000L
#define NWTC_REMPORT_MASK       0x0200L
#       define NWTC_SET_RP      0x00000200L
#       define NWTC_UNSET_RP    0x02000000L
```

A tcp channel is considered configured when one flag in each category has been selected. Thus one of **NWTC_EXCL**, **NWTC_SHARED** or **NWTC_COPY**, one of **NWTC_LP_UNSET**, **NWTC_LP_SET** or **NWTC_LP_SEL**, one of **NWTC_SET_RA** or **NWTC_UNSET_RA**, and one of **NWTC_SET_RP** or **NWTC_UNSET_RP**.

The acc flags control the access to a certain TCP port. **NWTC_EXCL** means exclusive access. An attempt to configure a channel will be denied if the same port is specified as that of a channel that requested exclusive access. **NWTC_SHARED** indicates that several channels use the same port but cooperate. If the shared mode is specified for one channel than all other channel that use the same port should also be configured with the **NWTC_SHARED** flag. **NWTC_COPY** is specified when the programmer does not care about other channels. This is the default.

The locport flags control which TCP port is used for communication. **NWTC_LP_UNSET** indicates the absence of a local port. This is the default. **NWTC_LP_SET** means that the **nwtc_locport** field contains the local port to be used by TCP. This value must be in network byte order (see **hton**(3).) **NWTC_LP_SEL** requests the TCP server to pick a port. This port will be in the range from 32768 to 65535 and will be unique.

The **remaddr** flags specify which hosts are acceptable for connections. **NWTC_SET_RA** indicates that only connection to the host specified in **nwtc_remaddr** are acceptable. **Nwtc_remaddr** should be in network byte order (see **hton**(3).) **NWTC_UNSET_RA** allows every host on the other side of a connection. This is the default.

The **remport** flags specify which remote ports are acceptable for connections. **NWTC_SET_RP** indicates that only the port specified in **nwtc_remport** is acceptable. **NWTC_UNSET_RP** allows every port on the other side of a connection. This is the default.

ioctl(*fd*, NWIOTCPSHUTDOWN)

The **NWIOTCPSHUTDOWN** tells the TCP/IP server that no more data will be sent over the channel specified by *fd*. This command can be issued when the channel is connected to a remote TCP/IP server. The TCP/IP server will tell the remote TCP/IP server and the client of the remote TCP/IP server will receive an end-of-file indication.

ioctl(*fd*, NWIOGTCPOPT, &struct nwio_tcpopt)
ioctl(*fd*, NWIOSTCPOPT, &struct nwio_tcpopt)

The behaviour of a TCP channel may be changed by setting a number of options. The TCP options can be obtained with the **NWIOGTCPOPT** ioctl and set with the **NWIOSTCPOPT** ioctl. The options are passed in a **struct nwio_tcpopt** as defined in <net/gen/tcp_io.h>:

```
typedef struct nwio_tcpopt
{
      u32_t nwto_flags;
} nwio_tcpconf_t;

#define NWTO_NOFLAG     0x0000L
#define NWTO_SND_URG_MASK       0x0001L
#       define NWTO_SND_URG     0x00000001L
#       define NWTO_SND_NOTURG  0x00010000L
#define NWTO_RCV_URG_MASK       0x0002L
#       define NWTO_RCV_URG     0x00000002L
#       define NWTO_RCV_NOTURG  0x00020000L
#define NWTO_BSD_URG_MASK       0x0004L
#       define NWTO_BSD_URG     0x00000004L
#define NWTO_DEL_RST_MASK       0x0008L
#       define NWTO_DEL_RST     0x00000008L
```

The **NWTO_SND_URG** option causes bytes written to the channel to be send out as urgent data. On receiving an **EURG** error the **NWTO_RCV_URG** option must be set to switch over to reading urgent data. When all urgent data has been read an **ENOURG** error will follow, indicating that the option must be cleared with **NWTO_RCV_NOTURG**. Alas the BSD implementation of urgent data disagrees with the RFC's, so to be BSD compatible one must set the **NWTO_BSD_URG** option beforehand on a channel that is to send or receive urgent data. Given that the BSD implementation is the regarded as the TCP/IP standard one should always use the BSD style. The **NWTO_DEL_RST** option delays a failure response on a connect to the same port as the current open connection. Without this option a connect would fail if a server is not yet listening. With this option a connect will linger on until the server starts listening. This option is useful for a server that opens a connection, tells the remote end the local port number and then listens (FTP), or for a program that forks off servers for incoming connections (TELNET). A new connection may come in before a new listen can be started, so it is nice if the new connect doesn't fail. Use this option only when it is clearly needed.

UDP Functions

ioctl(*fd*, NWIOGUDPOPT, &struct nwio_udpopt)

The **NWIOGUDPOPT** ioctl returns the current options that result from the default options and the options set with **NWIOSUDPOPT**. When **NWUO_LP_SEL** or **NWUO_LP_SET** is selected the local port is returned in **nwuo_locport**. When **NWUO_RP_SET** is selected the remote port is returned in **nwuo_remport**. The local address is always returned in **nwuo_locaddr**, and when **NWUO_RA_SET** is selected the remote address is returned in **nwuo_remaddr**.

ioctl(*fd*, NWIOSUDPOPT, &struct nwio_udpopt)

A UDP channel must be configured using the **NWIOSUDPOPT** ioctl before any data can be read or written. **NWIOSUDPOPT** takes two parameters, a file descriptor to an open UDP device and pointer to a **nwio_udpopt** structure that describes the requested configuration. The **nwio_udpopt** structure is defined in <net/gen/udp_io.h> as:

```
typedef struct nwio_udpopt
{
      unsigned long nwuo_flags;
      udpport_t nwuo_locport;
      udpport_t nwuo_remport;
      ipaddr_t nwuo_locaddr;
      ipaddr_t nwuo_remaddr;
} nwio_udpopt_t;

#define NWUO_NOFLAGS            0x0000L
#define NWUO_ACC_MASK           0x0003L
#define         NWUO_EXCL               0x00000001L
#define         NWUO_SHARED             0x00000002L
#define         NWUO_COPY               0x00000003L
#define NWUO_LOCPORT_MASK       0x000CL
#define         NWUO_LP_SEL             0x00000004L
#define         NWUO_LP_SET             0x00000008L
#define         NWUO_LP_ANY             0x0000000CL
#define NWUO_LOCADDR_MASK       0x0010L
#define         NWUO_EN_LOC             0x00000010L
#define         NWUO_DI_LOC             0x00100000L
#define NWUO_BROAD_MASK         0x0020L
#define         NWUO_EN_BROAD           0x00000020L
#define         NWUO_DI_BROAD           0x00200000L
#define NWUO_REMPORT_MASK       0x0100L
#define         NWUO_RP_SET             0x00000100L
#define         NWUO_RP_ANY             0x01000000L
#define NWUO_REMADDR_MASK       0x0200L
#define         NWUO_RA_SET             0x00000200L
#define         NWUO_RA_ANY             0x02000000L
#define NWUO_RW_MASK            0x1000L
#define         NWUO_RWDATONLY          0x00001000L
#define         NWUO_RWDATALL           0x10000000L
#define NWUO_IPOPT_MASK         0x2000L
#define         NWUO_EN_IPOPT           0x00002000L
#define         NWUO_DI_IPOPT           0x20000000L
```

A UDP channel is considered configured when one flag in each category has been selected. Thus one of **NWUO_EXCL**, **NWUO_SHARED** or **NWUO_COPY**, one of **NWUO_LP_SEL**, **NWUO_LP_SET** or **NWUO_LP_ANY**, one of **NWUO_EN_LOC** or **NWUO_DI_LOC**, one of **NWUO_EN_BROAD**, or **NWUO_DI_BROAD**, one of **NWUO_RP_SET**, or **NWUO_RP_ANY**, one of **NWUO_RA_SET**, or **NWUO_RA_ANY**, one of **NWUO_RWDATONLY**, or **NWUO_RWDATALL**, and one of **NWUO_EN_IPOPT**, or **NWUO_DI_IPOPT**. The acc flags control the access to a certain UDP port. **NWUO_EXCL** means exclusive access: no other channel can use this port. **NWUO_SHARED** means shared access: only channels that specify shared access can use this port and all packets that are received are handed to at most one channel. **NWUO_COPY** imposes no access restriction and all channels get a copy of every received packet for that port.

The **locport** flags control the selection of the UDP port for this channel. **NWUO_LP_SEL** requests the server to pick a port. This port will be in the range from 32768 to 65535 and it will be unique. **NWUO_LP_SET** sets the local port to the value of the **nwuo_locport** field. **NWUO_LP_ANY** does not select a port. Reception of data is therefore not possible but it is possible to send data. (See **tcpip_read**, and **tcpip_write** above).

The **locaddr** flags control the reception of packets. **NWUO_EN_LOC** enables the reception of packets with the local IP address as destination. **NWUO_DI_LOC** disables the reception of packet for the local IP address.

The **broad** flags control the reception of broadcast packets. **NWUO_EN_BROAD** enables the reception of broadcast packets and **NWUO_DI_BROAD** disables the reception of broadcast packets.

The **remport** flags let the client to specify one specific remote UDP port or to allow any remote port. **NWUO_RP_SET** sets the remote UDP port to the value of **nwuo_remport**. Only packets with a matching remote port will be delivered and all packets will be sent to that port. **NWUO_RP_ANY** allows reception of packets form any port and when transmitting packets the remote port has to be specified.

The **remaddr** flags control the remote IP address. **NWUO_RA_SET** sets the remote IP address the value of **nwuo_remaddr**. Only packets from that address will be delivered and all packets will be sent to that address. **NWUO_RA_ANY** allows reception of packets from any host and when transmitting packets the remote host has to be specified.

The **rw** flags control the format of the data to be sent or received. With **NWUO_RWDATONLY** only the data part of a UDP packet is sent to the server and only the data part is received from the server. The **NWUO_RWDATALL** mode presents the data part of a UDP packet with a header that contains the source and destination IP address, source and destination UDP ports, the IP options, etc. The server expects such a header in front of the data to be transmitted.

The **ipopt** flags control the delivery and transmission of IP options. When **NWUO_EN_IPOPT** is set IP, options will be delivered and sent. When **NWUO_DI_IPOPT** is set IP option will be stripped from received packets and no IP options will be sent.

FILES

/dev/eth[01]	First and second raw ethernet.
/dev/psip[01]	First and second Pseudo IP network.
/dev/ip[0123]	IP devices for two ethernets and two Pseudo IP networks.
/dev/tcp[0123]	TCP devices for same four networks.
/dev/udp[0123]	UDP devices.
/dev/eth, /dev/psip, /dev/ip, /dev/tcp, /dev/udp	Devices for the default network, links to the devices above. **Eth** is only present if ethernet is the default, **psip** only for pseudo IP.

SEE ALSO

hton(3), **oneC_sum**(3), **set_net_default**(8), **boot**(8).

DIAGNOSTICS

Several errors may be returned by the TCP/IP server. The error code is found in the **errno** variable if the **read**, **write**, or **ioctl** call returns -1. The TCP/IP error codes defined in <errno.h> are:

EPACKSIZE	This indicates an attempt to read or write with a buffer that is too large or too small.
EOUTOFBUFS	The TCP/IP server has insufficient memory to execute the request.
EBADIOCTL	This indicates an attempt to execute a command the particular server does not understand. For example, a **NWIOGTCPCONF** on an ETH channel.
EBADMODE	The request is refused because the channel is not fully configured, in the wrong state or the parameters are invalid.
EBADDEST	This indicates an illegal destination address for a packet.
EDSTNORCH	The destination is not reachable.
EISCONN	The channel is already connected so a second request is refused.
EADDRINUSE	This address is in use.
ECONNREFUSED	The connection is refused by the other side.
ECONNRESET	The connection is reset (non-gracefully terminated) by the other side.
ETIMEDOUT	The connection is terminated due to an expired timer.
EURG	Urgent data is present and the current receive mode does not allow urgent data to be transferred.
ENOURG	No urgent data is present and a request came for urgent data.
ENOTCONN	The request requires a connected channel and the channel is not connected.
ESHUTDOWN	The connection is shut down. That is, a **NWIOTCPSHUTDOWN** has been executed so no more data can be transmitted.
ENOCONN	The connection does not exist.
EGENERIC	A generic error code for extremely weird cases.

AUTHOR

Philip Homburg (philip@cs.vu.nl)

NAME

lp − line printer

DESCRIPTION

The **lp** device refers to the line printer attached to the parallel port. Any byte written to this device is printed. Only one process may have the device open.

The **write (2)** call may return with a smaller count then the number of bytes requested to write. The next write call is then likely to fail with the error code **EAGAIN** if the printer is out of paper, or **EIO** if the printer is turned off.

FILES

/dev/lp Parallel port device.

SEE ALSO

lp(1).

BUGS

Only one parallel port is supported.

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

mtio − magnetic tape commands

SYNOPSIS

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

DESCRIPTION

The magnetic tape devices described in **sd**(4) may be sent commands or queried for their status using the following ioctl calls:

ioctl(*fd*, **MTIOCTOP**, &**struct mtop**)
ioctl(*fd*, **MTIOCGET**, &**struct mtget**)

The struct mtop, struct mtget and associated definitions are defined in <sys/mtio.h> as follows:

```
/* Tape operations: ioctl(fd, MTIOCTOP, &struct mtop) */

struct mtop {
	short	mt_op;		/* Operation (MTWEOF, etc.) */
	int	mt_count;	/* Repeat count. */
};

#define MTWEOF	0	/* Write End-Of-File Marker */
#define MTFSF	1	/* Forward Space File mark */
#define MTBSF	2	/* Backward Space File mark */
#define MTFSR	3	/* Forward Space Record */
#define MTBSR	4	/* Backward Space Record */
#define MTREW	5	/* Rewind tape */
#define MTOFFL	6	/* Rewind and take Offline */
#define MTNOP	7	/* No-Operation, set status only */
#define MTRETEN	8	/* Retension (completely wind and rewind) */
#define MTERASE	9	/* Erase the tape and rewind */
#define MTEOM	10	/* Position at End-Of-Media */
#define MTMODE	11	/* Select tape density */
#define MTBLKZ	12	/* Select tape block size */

/* Tape status: ioctl(fd, MTIOCGET, &struct mtget) */

struct mtget {
	short	mt_type;	/* Type of tape device. */

	/* Device dependent "registers". */
	short	mt_dsreg;	/* Drive status register. */
	short	mt_erreg;	/* Error register. */

	/* Misc info. */
	off_t	mt_resid;	/* Residual count. */
	off_t	mt_fileno;	/* Current File Number. */
	off_t	mt_blkno;	/* Current Block Number within file. */
	off_t	mt_blksize;	/* Current block size. */
};
```

See **mt**(1) for a detailed description on what each operation does. The mt_type field is always zero, there is no use for it yet. Mt_dsreg is 0 (OK), 1 (Error), or 2 (EOF encountered.) Mt_erreg holds the SCSI sense key of the last operation. Mt_blksize is the current tape block size in bytes, zero if the block size is variable.

Note that one can issue these commands on a file descriptor that is in use to read or write data, something that **mt** can't do. So you can add eof markers in the middle of an output stream, or get the status of a device before a rewind-on-close tape rewinds.

The driver will automatically add an end of file marker to a tape that is written to if you execute a space command. If you write eof markers yourself then the driver will not add one extra on close.

SEE ALSO

mt(1), **sd**(4).

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

sd, st, sg − SCSI hard disk / tape / generic

DESCRIPTION

The **sd***, **st***, **sg*** family of devices refer to the SCSI hard disk, tape and generic driver using the Adaptec 154x series of controllers. This manual page only describes the differences between the sd and hd devices, read **hd**(4) first.

The devices numbers of the SCSI devices are statically mapped onto the SCSI targets 0 to 7. This is done like the hd devices with **sd[0−4]** referring to target 0, **sd[5−9]** to target 1, etc. The logical unit number is always 0, because devices with more than one logical unit are virtually extinct. The mapping may be changed from the boot environment however (see **boot**(8)).

Tapes start at minor device 64, with **nrst0** at minor 64, **rst0** at 65, **nrst1** at 66, etc. The mapping is again static to target (minor − 64) / 2. The **rst** devices rewind the tape on close, the **nrst** devices do not. See **mt**(1), and **mtio**(4) for a description of the commands that may be sent to the tape, either from the command prompt or from a program.

Through the eight raw generic devices **rsg[0−7]** starting at minor 120, one can send SCSI commands to any SCSI device from user mode. (Minix-vmd only.)

The driver returns a drive geometry of 64 heads by 32 sectors per track for small disks with the DIOCGETP ioctl. For large disks 255x63 is returned. The size in sectors is usually larger than the largest cylinder number indicates, because the disk is not likely to exactly match that faked geometry. Note that DOS may not be able to access those last few sectors.

Disk like devices.

Removable disks (floppies), CD-ROM’s and WORM disks may also be accessed through the **sd** devices. One is not allowed to write a WORM disk however, because it is likely to be taken from an alien operating system, so it seems safer to not allow Minix to stomp over it. One usually needs special O.S. support to keep one from writing to the same block twice.

The **DIOCEJECT** ioctl ejects CD-ROMs, floppies, etc. (See **eject**(1).) A fixed disk spins down if it supports the stop command.

SCSI Tapes

There are two types of SCSI tapes drives supported by the driver: fixed or variable block size tape drives. Examples of the first kind are cartridge tapes, with a fixed 512 bytes block size. An Exabyte tape drive has a variable block size, with a minimum of 1 byte and a maximum of 245760 bytes (see the documentation of such devices.) The maximum is truncated to 32767 bytes for Minix-86 and 61440 bytes for Minix-vmd, because the driver can’t move more bytes in a single request.

A read or write to a fixed block size tape must be a precise multiple of the block size, any other count gives results in an I/O error. A read from a variable block sized tape must be large enough to accept the block that is read, otherwise an I/O error will be returned. A write can be any size above the minimum, creating a block of that size. If the write count is larger than the maximum block size then more blocks are written until the count becomes zero. The last block must be larger than the minimum of course. (This minimum is often as small as 1 byte, as for the Exabyte.)

The **mt blksize** command may be used to select a fixed block size for a variable block sized tape. This will speed up I/O considerably for small block sizes. (Some systems can only use fixed mode and will write an Exabyte tape with 1024 byte blocks, which read very slow in variable mode.)

A tape is a sequence of blocks and filemarks. A tape may be opened and blocks may be read from it upto a filemark, after that all further reads return 0. After the tape is closed and reopened one can read the blocks following the filemark if using a non-rewinding device. This makes the tape look like a sequence of files.

If a tape has been written to or opened in write-only mode, then a filemark is written if the tape is closed or if a space command is issued. No extra filemark is written if the drive is instructed to write filemarks.

Raw Generic Devices

Under Minix-vmd one can use the generic SCSI devices to program a SCSI device entirely from user mode. The disk and tape devices probe for devices when opened, start disks and load tapes, but the generic devices do nothing of this. Given an open file descriptor to any SCSI character device (not just the generic devices) one can use the following ioctl:

```
ioctl(fd, SCIOCCMD, &scsicmd)
```

The structure whose address is passed as the third argument is defined in <sys/scsi.h> as follows:

```
struct scsicmd {
        void     *cmd;
        size_t   cmdlen;
        void     *buf;
        size_t   buflen;
        void     *sense;
        size_t   senselen;
        int      dir;
};
```

Cmd and **cmdlen** hold the address and length of an object holding a Group 0 or Group 1 SCSI command. The next two fields describe a buffer of at most 8 kilobytes used in the data in or out phase. **Dir** is 0 if data is to be read from the device, 1 if data is written to the device. If the ioctl succeeds then 0 is returned, otherwise -1 with **errno** set to **EIO** and the request sense info returned in the buffer described by the sense and senselen fields. If the sense key is zero on error then a host adapter error occurred, this means that the device is most likely turned off or not present.

FILES

/dev/sd[0−9], /dev/sd[1−46−9][a−d]	Usual disk devices.
/dev/rst4, /dev/nrst4	Usual tape device.
/dev/rsg[0−7]	Raw generic devices.

SEE ALSO

hd(4), **mt**(1), **eject**(1), **mtio**(4), **dd**(1).

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

tty, termios − terminals

DESCRIPTION

The **tty** driver family takes care of all user input and output. It governs the keyboard, the console, the serial lines, and pseudo ttys. Input on any of these devices undergoes "input processing", and output undergoes "output processing" according to the standard termios terminal interface.

Input processing

Each terminal device has an input queue. This queue is used to store preprocessed input characters, and to perform the backspacing and erase functions. Some special characters like a newline make the contents of the queue available to a process reading from the terminal. Characters up to and including the newline, or another so-called "line break", may be read by a process. The process need not read all characters at once. An input line may be read byte by byte if one wants to. A line break just makes characters available for reading, thats all.

When data is made available depends on whether the tty is in canonical mode or not. In canonical mode the terminal processes input line by line. A line ends with a newline (**NL**), end-of-file (**EOF**), or end-of-line (**EOL**). Characters that have not been delimited by such a line break may be erased one by one with the **ERASE** character or all at once with the **KILL** character. Once a line break is typed the characters become available to a reading process and can no longer be erased. Once read they are removed from the input queue. Several lines may be gathered in the input queue if no reader is present to read them, but a new reader will only receive one line. Two line breaks are never returned in one read call. The input queue has a maximum length of **MAX_CANON** characters. Any more characters are discarded. One must use **ERASE** or **KILL** to make the terminal functioning again if the input queue fills up. If nonblocking I/O is set then −1 is returned with **errno** set to **EAGAIN** if the reader would otherwise be blocked.

In non-canonical mode (raw mode for short) all characters are immediately available to the reader in principle. One may however tune the terminal to bursty input with the **MIN** and **TIME** parameters, see the raw I/O parameters section below. In raw mode no characters are discarded if the input queue threatens to overflow if the device supports flow control.

Output processing

Characters written to a terminal device may undergo output processing, which is usually just inserting a carriage returns before newlines. A writer may return before all characters are output if the characters can be stored in the output buffers. If not then the writer may be blocked until space is available. If non-blocking I/O is set then only the count of the number of bytes that can be processed immediately is returned. If no characters can be written at all then −1 is returned with **errno** set to **EAGAIN**.

Special characters

Some characters have special functions in some of the terminal modes. These characters are as follows, with the Minix defaults shown in parentheses:

INTR (^?)
: Special input character that is recognized if **ISIG** is set. (For **ISIG** and other flags see the various modes sections below.) It causes a **SIGINT** signal to be sent to all processes in the terminal process group. (See the section on session leaders below.)

QUIT (^\)
: Special input character if **ISIG** is set. Causes a **SIGQUIT** signal to be sent to the terminal process group.

ERASE (^H)
: Special input character if **ICANON** is set. Erases the last character in the current line.

KILL (^U)
: Special input character if **ICANON** is set. Erases the entire line.

EOF (^D)
: Special input character if **ICANON** is set. It is a line break character that is not itself returned to a reader. If EOF is typed with no input present then the read returns zero, which normally causes

the reader to assume that end-of-file is reached.

CR (ˆM)
: Special input character if **IGNCR** or **ICRNL** is set. It is a carriage return ('\r'). If **IGNCR** is set then **CR** is discarded. If **ICRNL** is set and **IGNCR** is not set then **CR** is changed into an **NL** and has the same function as **NL.**

NL (ˆJ)
: Special input character if **ICANON** is set. It is both a newline ('\n') and a line break.
Special output character if **OPOST** and **ONLCR** are set. A **CR NL** sequence is output instead of just **NL**. (Minix specific, but almost mandatory on any UNIX-like system.)

TAB (ˆI)
: Special character on output if **OPOST** and **XTABS** are set. It is transformed into the number of spaces necessary to reach a column position that is a multiple of eight. (Only needed for terminals without hardware tabs.)

EOL (undefined)
: Special input character if **ICANON** is set. It is an additional line break.

SUSP (ˆZ)
: Special input character if job control is implemented and **ISIG** is set. It causes a **SIGTSTP** signal to be send to the terminal process group. (Minix does not have job control.)

STOP (ˆS)
: Special input character if **IXON** is set. It suspends terminal output and is then discarded.

START (ˆQ)
: Special output character if **IXON** is set. It starts terminal output if suspended and is then discarded. If **IXANY** is also set then any other character also starts terminal output, but they are not discarded.

REPRINT (ˆR)
: Special input character if **IEXTEN** and **ECHO** are set. Reprints the input queue from the last line break onwards. A reprint also happens automatically if the echoed input has been messed up by other output and **ERASE** is typed.

LNEXT (ˆV)
: Special input character if **IEXTEN** is set. It is the "literal next" character that causes the next character to be input without any special processing.

DISCARD (ˆO)
: Special input character if **IEXTEN** is set. Causes output to be discarded until it is typed again. (Implemented only under Minix-vmd.)

All of these characters except **CR**, **NL** and **TAB** may be changed or disabled under Minix. (Changes to **START** and **STOP** may be ignored under other termios implementations.) The **REPRINT** and **LNEXT** characters are Minix extensions that are commonly present in other implementations. POSIX is unclear on whether **IEXTEN, IGNCR** and **ICRNL** should be active in non-canonical mode, but under Minix they are.

Terminal attributes

The attributes of a terminal, such as whether the mode should be canonical or non-canonical, are controlled by routines that use the **termios** structure as defined in **<termios.h>**:

```
struct termios {
      tcflag_t   c_iflag;        /* input modes */
      tcflag_t   c_oflag;        /* output modes */
      tcflag_t   c_cflag;        /* control modes */
      tcflag_t   c_lflag;        /* local modes */
      speed_t    c_ispeed;       /* input speed */
      speed_t    c_ospeed;       /* output speed */
      cc_t       c_cc[NCCS];     /* control characters */
```

```
};
```

The types **tcflag**, **speed_t** and **cc_t** are defined in **<termios.h>** as unsigned integral types.

Input Modes

The **c_iflag** field contains the following single bit flags that control input processing:

ICRNL
Map **CR** to **NL** on input.

IGNCR
Ignore **CR** on input. This flag overrides **ICRNL**.

INLCR
Map **NL** to **CR** on input. This is done after the **IGNCR** check.

IXON
Enable start/stop output control.

IXOFF
Enable start/stop input control. (Not implemented.)

IXANY
Allow any character to restart output. (Minix specific.)

ISTRIP
Strip characters to seven bits.

IGNPAR
Ignore characters with parity errors. (Not implemented.)

INPCK
Enable input parity checking. (Not implemented.)

PARMRK
Mark parity errors by preceding the faulty character with '\377', '\0'. The character '\377' is preceded by another '\377' to avoid ambiguity. (Not implemented.)

BRKINT
Send the signal **SIGINT** to the terminal process group when receiving a break condition. (Not implemented.)

IGNBRK
Ignore break condition. If neither **BRKINT** or **IGNBRK** is set a break is input as a single '\0', or if **PARMRK** is set as '\377', '\0', '\0'. (Breaks are always ignored.)

Output Modes

The **c_oflag** field contains the following single bit flags that control output processing:

OPOST
Perform output processing. This flag is the "main switch" on output processing. All other flags are Minix specific.

ONLCR
Transform an **NL** to a **CR NL** sequence on output. Note that a key labeled "RETURN" or "ENTER" usually sends a **CR**. In line oriented mode this is normally transformed into **NL** by **ICRNL**. **NL** is the normal UNIX line delimiter ('\n'). On output an **NL** is transformed into the **CR NL** sequence that is necessary to reach the first column of the next line. (This is a common output processing function for UNIX-like systems, but not always separately switchable by an **ONLCR** flag.)

XTABS
Transform a **TAB** into the number of spaces necessary to reach a column position that is a multiple of eight.

ONOEOT
Discard **EOT** (^D) characters. (Minix-vmd only.)

Control Modes

The **c_cflag** field contains the following single bit flags and bit field for basic hardware control:

CLOCAL
Ignore modem status lines.

CREAD
Enable receiver. (The receiver is always enabled.)

CSIZE Number of bits per byte. **CSIZE** masks off the values **CS5**, **CS6**, **CS7** and **CS8** that indicate that 5, 6, 7 or 8 bits are used.

CSTOPB
Send two stop bits instead of one. Two stop bits are normally used at 110 baud or less.

PARENB
Enable parity generation.

PARODD
Generate odd parity if parity is generated, otherwise even parity.

HUPCL
Drop the modem control lines on the last close of the terminal line. (Not implemented.)

Local Modes

The **c_lflag** field contains the following single bit flags that control various functions:

ECHO Enable echoing of input characters. Most input characters are echoed as they are. Control characters are echoed as ^*X* where *X* is the letter used to say that the control character is **CTRL−***X*. The **CR**, **NL** and **TAB** characters are echoed with their normal effect unless they are escaped by **LNEXT**.

ECHOE
If **ICANON** and **ECHO** are set then echo **ERASE** and **KILL** as one or more backspace-space-backspace sequences to wipe out the last character or the entire line, otherwise they are echoed as they are.

ECHOK
If **ICANON** and **ECHO** are set and **ECHOE** is not set then output an **NL** after the **KILL** character. (For hardcopy terminals it is best to unset **ECHOE** and to set **ECHOK**.)

ECHONL
Echo **NL** even if **ECHO** is not set, but **ICANON** is set.

ICANON
Canonical input. This enables line oriented input and erase and kill processing.

IEXTEN
Enable implementation defined input extensions.

ISIG Enable the signal characters **INTR**, **QUIT** and **SUSP**.

NOFLSH
Disable the flushing of the input and output queues that is normally done if a signal is sent.

TOSTOP
Send a **SIGTTOU** signal if job control is implemented and a background process tries to write. (Minix has no job control.)

Input and output speed

The input and output speed are encoded into the **c_ispeed** and **c_ospeed** fields. **<termios.h>** defines the symbols **B0**, **B50**, **B75**, **B110**, **B134**, **B150**, **B200**, **B300**, **B600**, **B1200**, **B1800**, **B2400**, **B4800**, **B9600**, **B19200**, **B38400**, **B57600** and **B115200** as values used to indicate the given baud rates. The zero baud rate, **B0**, if used for the input speed causes the input speed to be equal to the output speed. Setting the output speed to zero hangs up the line. One should use the functions **cfgetispeed()**, **cfgetospeed()**, **cfsetispeed()** and **cfsetospeed()** to get or set a speed, because the **c_ispeed** and

c_ospeed fields may not be visible under other implementations. (The **c_ispeed** and **c_ospeed** fields and the **B57600** and **B115200** symbols are Minix specific.)

Special characters

The **c_cc** array contains the special characters that can be modified. The array has length **NCCS** and is subscripted by the symbols **VEOF**, **VEOL**, **VERASE**, **VINTR**, **VKILL**, **VMIN**, **VQUIT**, **VTIME**, **VSUSP**, **VSTART**, **VSTOP**, **VREPRINT**, **VLNEXT** and **VDISCARD**. All these symbols are defined in **<termios.h>**. Some implementations may give the same values to the **VMIN** and **VTIME** subscripts and the **VEOF** and **VEOL** subscripts respectively, and may ignore changes to **START** and **STOP**. (Under Minix all special characters have there own *c_cc* slot and can all be modified.)

Raw I/O Parameters

The **MIN** and **TIME** parameters can be used to adjust a raw connection to bursty input. **MIN** represents a minimum number of bytes that must be received before a read call returns. **TIME** is a timer of 0.1 second granularity that can be used to time out a read. Setting either of these parameters to zero has special meaning, which leads to the following four possibilities:

MIN > 0, TIME > 0

TIME is an inter-byte timer that is started (and restarted) when a byte is received. A read succeeds when either the minimum number of characters is received or the timer expires. Note that the timer starts **after** the first character, so the read returns at least one byte.

MIN > 0, TIME = 0

Now the timer is disabled, and a reader blocks indefinitely until at least **MIN** characters are received.

MIN = 0, TIME > 0

TIME is now a read timer that is started when a read is executed. The read will return if the read timer expires or if at least one byte is input. (Note that a value of zero may be returned to the reader.)

MIN = 0, TIME = 0

The bytes currently available are returned. Zero is returned if no bytes are available.

User Level Functions

Termios attributes are set or examined, and special functions can be performed by using the functions described in **termios**(2).

Session Leaders and Process Groups

With the use of the **setsid()** function can a process become a session leader. A session leader forms a process group with a process group id equal to the process id of the session leader. If a session leader opens a terminal device file then this terminal becomes the controlling tty of the session leader. Unless the terminal is already the controlling tty of another process, or unless the **O_NOCTTY** flag is used to prevent the allocation of a controlling tty. The process group of the session leader is now remembered as the terminal process group for signals sent by the terminal driver. All the children and grandchildren of the session leader inherit the controlling terminal and process group until they themselves use **setsid()**.

The controlling tty becomes inaccessible to the children of the session leader when the session leader exits, and a hangup signal is sent to all the members of the process group. The input and output queues are flushed on the last close of a terminal and all attributes are reset to the default state.

A special device **/dev/tty** is a synonym for the controlling tty of a process. It allows a process to reach the terminal even when standard input, output and error are redirected. Opening this device can also be used as a test to see if a process has a controlling tty or not.

For Minix a special write-only device **/dev/log** exists for processes that want to write messages to the system console. Unlike the console this device is still accessible when a session leader exits.

Minix-vmd also has a **/dev/log** device, but this device is read-write. All messages written to the log device or to the console when X11 is active can be read from **/dev/log**. The system tries to preserve the log buffer over a reboot so that panic messages reappear in the log if the system happens to crash.

Pseudo Terminals

Pseudo ttys allow a process such as a remote login daemon to set up a terminal for a remote login session. The login session uses a device like **/dev/ttyp0** for input and output, and the remote login daemon uses the device **/dev/ptyp0** to supply input to or take output from the login session and transfer this to or from the originating system. So the character flow may be: Local user input sent to the remote system is written to **/dev/ptyp0** by the remote login daemon, undergoes input processing and appears on **/dev/ttyp0** as input to the login session. Output from the login session to **/dev/ttyp0** undergoes output processing, is read from **/dev/ptyp0** by the remote login daemon and is send over to the local system to be displayed for the user. (So there are only four data streams to worry about in a pseudo terminal.)

A pseudo terminal can be allocated by trying to open all the controlling devices **/dev/pty***nn* one by one until it succeeds. Further opens will fail once a pty is open. The process should now fork, the child should become session leader, open the tty side of the pty and start a login session.

If the tty side is eventually closed down then reads from the pty side will return zero and writes return −1 with **errno** set to **EIO**. If the pty side is closed first then a **SIGHUP** signal is sent to the session leader and further reads from the tty side return zero and writes return −1 with **errno** set to **EIO**. (Special note: A line erase may cause up to three times the size of the tty input queue to be sent to the pty reader as backspace overstrikes. Some of this output may get lost if the pty reader cannot accept it all at once in a single read call.)

Backwards compatibility

The **TIOCGETP**, **TIOCSETP**, **TIOCGETC** and **TIOCSETC** ioctl functions that are used by the old **sgtty** terminal interface are still supported by the terminal driver by emulation. Note that these old functions cannot control all termios attributes, so the terminal must be in a relatively sane state to avoid problems.

FILES

The list below shows all devices that Minix and Minix-vmd have. Not all of these devices are configured in by default, as indicated by the numbers (i/j/k, l/m/n) that tell the minimum, default and maximum possible number of these devices for Minix (i/j/k) and Minix-vmd (l/m/n).

/dev/console	System console.
/dev/ttyc[1-7]	Virtual consoles. (0/1/7, 0/1/7)
/dev/log	Console log device.
/dev/tty0[0-3]	Serial lines. (0/2/2, 4/4/4)
/dev/tty[p-w][0-f]	Pseudo ttys. (0/0/64, 1/32/128)
/dev/pty[p-w][0-f]	Associated pseudo tty controllers.

SEE ALSO

stty(1), **termios**(3), **setsid**(2), **read**(2), **write**(2).

BUGS

A fair number of flags are not implemented under Minix (yet). Luckily they are very limited utility and only apply to RS-232, not to the user interface.

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)