ULTRIX

digital

Reference Pages
Section 4: Special Files

Order Number: AA–LY17B–TE

ULTRIX

Reference Pages Section 4: Special Files

Order Number: AA-LY17B-TE
June 1990

Product Version: ULTRIX Version 4.0 or higher

This manual describes special files, related device driver functions, databases, and network support for both RISC and VAX platforms.

digital equipment corporation
maynard, massachusetts

digital	DECUS	ULTRIX Worksystem Software
	DECwindows	UNIBUS
CDA	DTIF	VAX
DDIF	MASSBUS	VAXstation
DDIS	MicroVAX	VMS
DEC	Q-bus	VMS/ULTRIX Connection
DECnet	ULTRIX	VT
DECstation	ULTRIX Mail Connection	XUI

About Reference Pages

The *ULTRIX Reference Pages* describe commands, system calls, routines, file formats, and special files for RISC and VAX platforms.

Sections

The reference pages are divided into eight sections according to topic. Within each section, the reference pages are organized alphabetically by title, except Section 3, which is divided into subsections. Each section and most subsections have an introductory reference page called `intro` that describes the organization and anything unique to that section.

Some reference pages carry a one- to three-letter suffix after the section number, for example, `scan`(1mh). The suffix indicates that there is a ''family'' of reference pages for that utility or feature. The Section 3 subsections all use suffixes and other sections may also have suffixes.

Following are the sections that make up the *ULTRIX Reference Pages*.

Section 1: Commands

This section describes commands that are available to all ULTRIX users. Section 1 is split between two binders. The first binder contains reference pages for titles that fall between A and L. The second binder contains reference pages for titles that fall between M and Z.

Section 2: System Calls

This section defines system calls (entries into the ULTRIX kernel) that are used by all programmers. The introduction to Section 2, `intro`(2), lists error numbers with brief descriptions of their meanings. The introduction also defines many of the terms used in this section.

Section 3: Routines

This section describes the routines available in ULTRIX libraries. Routines are sometimes referred to as subroutines or functions.

Section 4: Special Files

This section describes special files, related device driver functions, databases, and network support.

Section 5: File Formats

This section describes the format of system files and how the files are used. The files described include assembler and link editor output, system accounting, and file system formats.

Section 6: Games

The reference pages in this section describe the games that are available in the unsupported software subset. The reference pages for games are in the document *Reference Pages for Unsupported Software.*

Section 7: Macro Packages and Conventions

This section contains miscellaneous information, including ASCII character codes, mail addressing formats, text formatting macros, and a description of the root file system.

Section 8: Maintenance

This section describes commands for system operation and maintenance.

Platform Labels

The *ULTRIX Reference Pages* contain entries for both RISC and VAX platforms. Pages that have no platform label beside the title apply to both platforms. Reference pages that apply only to RISC platforms have a ''RISC'' label beside the title and the VAX-only reference pages that apply only to VAX platforms are likewise labeled with ''VAX.'' If each platform has the same command, system call, routine, file format, or special file, but functions differently on the different platforms, both reference pages are included, with the RISC page first.

Reference Page Format

Each reference page follows the same general format. Common to all reference pages is a title consisting of the name of a command or a descriptive title, followed by a section number; for example, `date(1)`. This title is used throughout the documentation set.

The headings in each reference page provide specific information. The standard headings are:

Name	Provides the name of the entry and gives a short description.
Syntax	Describes the command syntax or the routine definition. Section 5 reference pages do not use the Syntax heading.
Description	Provides a detailed description of the entry's features, usage, and syntax variations.
Options	Describes the command-line options.
Restrictions	Describes limitations or restrictions on the use of a command or routine.
Examples	Provides examples of how a command or routine is used.

Return Values	Describes the values returned by a system call or routine. Used in Sections 2 and 3 only.
Diagnostics	Describes diagnostic and error messages that can appear.
Files	Lists related files that are either a part of the command or used during execution.
Environment	Describes the operation of the system call or routine when compiled in the POSIX and SYSTEM V environments. If the environment has no effect on the operation, this heading is not used. Used in Sections 2 and 3 only.
See Also	Lists related reference pages and documents in the ULTRIX documentation set.

Conventions

The following documentation conventions are used in the reference pages.

%	The default user prompt is your system name followed by a right angle bracket. In this manual, a percent sign (%) is used to represent this prompt.
#	A number sign is the default superuser prompt.
`user input`	This bold typeface is used in interactive examples to indicate typed user input.
`system output`	This typeface is used in text to indicate the exact name of a command, routine, partition, pathname, directory, or file. This typeface is also used in interactive examples to indicate system output and in code examples and other screen displays.
UPPERCASE lowercase	The ULTRIX system differentiates between lowercase and uppercase characters. Literal strings that appear in text, examples, syntax descriptions, and function definitions must be typed exactly as shown.
rlogin	This typeface is used for command names in the Syntax portion of the reference page to indicate that the command is entered exactly as shown. Options for commands are shown in bold wherever they appear.
filename	In examples, syntax descriptions, and routine definitions, italics are used to indicate variable values. In text, italics are used to give references to other documents.
[ ]	In syntax descriptions and routine definitions, brackets indicate items that are optional.
{ | }	In syntax descriptions and routine definitions, braces enclose lists from which one item must be chosen. Vertical bars are used to separate items.

. . .	In syntax descriptions and routine definitions, a horizontal ellipsis indicates that the preceding item can be repeated one or more times.
. . .	A vertical ellipsis indicates that a portion of an example that would normally be present is not shown.
`cat(1)`	Cross-references to the *ULTRIX Reference Pages* include the appropriate section number in parentheses. For example, a reference to `cat`(1) indicates that you can find the material on the `cat` command in Section 1 of the reference pages.

Online Reference Pages

The ULTRIX reference pages are available online if installed by your system administrator. The `man` command is used to display the reference pages as follows:

To display the `ls`(1) reference page:

```
% man ls
```

To display the `passwd`(1) reference page:

```
% man passwd
```

To display the `passwd`(5) reference page:

```
% man 5 passwd
```

To display the Name lines of all reference pages that contain the word ''passwd'':

```
% man -k passwd
```

To display the introductory reference page for the family of 3xti reference pages:

```
% man 3xti intro
```

Users on ULTRIX workstations can display the reference pages using the unsupported `xman` utility if installed. See the `xman`(1X) reference page for details.

Reference Pages for Unsupported Software

The reference pages for the optionally installed, unsupported ULTRIX software are in the document *Reference Pages for Unsupported Software*.

Name

intro – introduction to special files

Description

Section 4 describes the special files, related driver functions, and networking support available in the system. In this part of the manual, the Syntax heading of each configurable device gives a sample specification for use in constructing a system description for the `config`(8) program. The Diagnostics heading lists messages that may appear on the console and in the system error log file `/usr/adm/syserr/syserr.<hostname>` due to errors in device operation.

This section contains descriptions of both configurable devices, 4 entries, and network-related device information, 4n, 4p, and 4f entries. The networking support is introduced in `intro`(4n). Software support for these devices comes in two forms. A hardware device may be supported with a character or block ''device driver'', or it may be used within the networking subsystem and have a ''network interface driver''.

There are some devices that have fixed input/output space CSR addresses and interrupt vector addresses. There are no address switches or jumpers that need to be set up by the customer or Field Service.

Files

`/dev/*`

See Also

intro(4n), MAKEDEV(8)

Name

networking – introduction to networking facilities

Syntax

#include <sys/socket.h>
#include <net/route.h>
#include <net/if.h>

Description

This section briefly describes the networking facilities available in the system. Documentation in this part of Section 4 is broken up into three areas: protocol families, protocols, and ''network interfaces''. Entries describing a protocol family are marked ''4f'', while entries describing protocol use are marked ''4p''. Hardware support for network interfaces is found among the standard ''4'' entries.

All network protocols are associated with a specific protocol family. A protocol family provides basic services to the protocol implementation to allow it to function within a specific network environment. These services can include packet fragmentation and reassembly, routing, addressing, and basic transport. A protocol family can support multiple methods of addressing, though the current protocol implementations do not. A protocol family normally comprises a number of protocols, one per socket type. It is not required that a protocol family support all socket types. A protocol family can contain multiple protocols supporting the same socket abstraction.

A protocol supports one of the socket abstractions detailed in `socket`(2). A specific protocol can be accessed either by creating a socket of the appropriate type and protocol family or by requesting the protocol explicitly when creating a socket. Protocols normally accept only one type of address format, usually determined by the addressing structure inherent in the design of the protocol family/network architecture. Certain semantics of the basic socket abstractions are protocol-specific. All protocols are expected to support the basic model for their particular socket type, but may, in addition, provide nonstandard facilities or extensions to a mechanism. For example, a protocol supporting the SOCK_STREAM abstraction may allow more than one byte of out-of-band data to be transmitted per out-of-band message.

A network interface is similar to a device interface. Network interfaces make up the lowest layer of the networking subsystem, interacting with the actual transport hardware. An interface may support one or more protocol families or address formats. The SYNTAX section of each network interface entry gives a sample specification of the related drivers for use in providing a system description to `config`(8.) The DIAGNOSTICS section lists messages that may appear on the console and in the system error log file `/usr/adm/syserr/syserr.<hostname>` due to errors in device operation.

Addressing

Associated with each protocol family is an address format. The following address formats are used by the system:

```
#define AF_UNIX    1  /* local to host (pipes, portals) */
#define AF_INET    2  /* internetwork: UDP, TCP, etc. */
#define AF_IMPLINK 3  /* arpanet imp addresses */
```

Routing

The network facilities provide limited packet routing. A simple set of data structures make up 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. Routing table manipulations can only be carried out by superuser.

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  rtentry *rt_next;
        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 types: for a specific host, for all hosts on a specific network, and 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 cannot 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 nonzero), the resources associated with it are not reclaimed until further references to it are released.

The routing code returns EEXIST if requested to duplicate an existing entry, ESRCH if requested to delete a nonexistent 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.

Interfaces

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

At boot time, each interface that has 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 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 that provide dynamic network-link layer address mapping facilities (for example, 10Mb/s Ethernets), 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 *ifrequest* 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 /* 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.

SIOCSTATE
: Read or set ownership and state of a device.

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 SIOCSIFBRDADDR) 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.

SIOCGIFNETMASK
Get network address mask.

SIOCSIFNETMASK
Set network address mask.

SIOCGIFBRDADDR
Get broadcast address associated with network interface.

SIOCSIFBRDADDR
Set broadcast address associated with network interface.

```
/*
 * 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 */
};
```

The following is the structure used in an SIOCSTATE request to set device state and ownership.

```
struct ifstate {
 char    ifr_name[IFNAMSIZ]; /* if name, e.g. "dmv0" */
 u_short if_family;          /* current family ownership */
 u_short if_next_family;     /* next family ownership */
 u_short if_mode:3,          /* mode of device */
         if_ustate:1,        /* user requested state */
         if_nomuxhdr:1,      /* if set, omit mux header */
         if_dstate:4,        /* current state of device */
         if_xferctl:1,       /* xfer control to nxt family */
         if_rdstate:1,       /* read current state */
         if_wrstate:1        /* set current state */
         if_reserved:4;
};
```

See Also

socket(2), ioctl(2), intro(4), config(8), routed(8c)

arp (4p)

Name

arp – Address Resolution Protocol

Syntax

pseudo-device ether

Description

The ARP protocol is used to map dynamically between DARPA Internet and 10Mb/s Ethernet addresses. It is used by all the 10Mb/s Ethernet interface drivers.

The ARP protocol caches Internet-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 messages are transmitted. The ARP protocol queues only the most recently ''transmitted'' packet while waiting for a mapping request to be responded to.

To enable communications with systems which do not use ARP, ioctls are provided to enter and delete entries in the Internet-to-Ethernet tables. The usage is:

```
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <net/if.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 ioctls may be applied to any socket descriptor *s*, but only by the superuser. 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  2   /* completed entry (arp_ha valid) */
#define      ATF_PERM 4   /* permanent entry */
#define      ATF_PUBL 8   /* publish (respond for other 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 that can be written are ATF_PERM and ATF_PUBL. ATF_PERM causes the entry to be permanent if the ioctl call succeeds. The ioctl may fail if more than four permanent Internet host 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 lets a SUN act as an ARP server, which can be used to make an ARP-only machine talk to a non-ARP machine.

The ARP protocol watches passively for a host that responds to an ARP mapping request for the local host's address.

Restrictions

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

Diagnostics

duplicate IP address!! sent from Ethernet address: %x:%x:%x:%x:%x:%x
ARP has discovered another host on the local network that responds to mapping requests for its own Internet address.

See Also

inet(4f), arp(8c), ifconfig(8c)

audit(4)

Name

audit – audit log interface

Description

This is a special character device that provides an interface for the audit daemon process, `/etc/sec/auditd`, to the kernel audit buffers.

Restrictions

This device should be readable and writable only by root, to protect access by nonsystem processes. The major number assigned to this device must correlate with the corresponding major number designation in the system kernel.

Files

`/dev/audit`

See Also

MAKEDEV(8)

Name

autoconf – diagnostics from the autoconfiguration code

Description

When ULTRIX bootstraps, it probes the machine it is running on and locates controllers, drives, and other devices, printing out what it finds on the console. This procedure is driven by a system configuration table, which is processed by `config`(8) and compiled into each kernel.

Diagnostics

cpu%d (version %f, implementation %d
A cpu is present.

fpu%d (version %f, implementation %d
An fpu is present.

pm%d at ibus%d
A monochrome or color graphics device is present.

cfb%d at ibus%d
A color graphic device is present.

dc%d at ibus%d
A serial line controller is present.

sii%d at ibus%d
An SCSI sii controller is present.

asc%d at ibus%d
An SCSI asc controller is present.

rz%d at sii%d/asc%d slave %d (
An SCSI disk device is present.

tz%d at sii%d/asc%d slave %d (
An SCSI tape device is present.

ln%d at ibus%d
An Ethernet interface is present.

See Also

intro(4), config(8)

cfb(4)

Name

cfb – color bitmap graphics

Syntax

device cfb0 at ibus? vector cfbvint

Description

The video subsystem provides a half page or full page, user-accessible bitmap display for graphics. The subsystem consists of a 1 Mbyte (color) block of dual port RAM, a mouse or tablet, a keyboard, and a video monitor.

The subsystem device driver supports a hybrid terminal with three minor devices. The first minor device emulates a glass tty with a screen that appears as an 80-column by 56-row page that scrolls from the bottom. This device is capable of being configured as the system console.

The second minor device is reserved for the mouse. This device is a source of mouse state changes. (A state change is defined as an X/Y axis mouse movement or button change.) When opened, the driver couples movements of the mouse with the cursor. Mouse position changes are filtered and translated into cursor position changes in an exponential manner. Rapid movements result in large cursor position changes. All cursor positions are range-checked to ensure that the cursor remains on the display.

The third minor device provides an access path for console output that does not disturb the graphics display. The caller can open the device `/dev/xcons`. When this device is open, the graphics driver redirects console device output to the input buffer of this device. This mechanism disables console output on the screen and saves the output for later display. This action preserves the graphic display integrity.

Input and output on the first and third minor devices are processed by the standard line disciplines.

The Hold Screen key is supported. The graphics driver treats this key as if CTRL/S or CTRL/Q has been entered. Pressing the Hold Screen key suspends the output (if it is not already suspended). To resume the output, press the Hold Screen key again.

Files

`/dev/console`	Console terminal or graphics device
`/dev/mouse`	Mouse or tablet graphics device
`/dev/xcons`	Console message window for workstation

See Also

console(4), devio(4), tty(4), ttys(5), MAKEDEV(8)

Name

cfl – RX01 console interface

Description

This is a simple interface to the RX01 floppy disk unit, which is part of the console LSI-11 subsystem for the VAX-11/780 (VAX-11/785). Access is given to the entire floppy, consisting of 77 tracks of 26 sectors of 128 bytes.

All I/O is raw; the seek addresses in raw transfers should be a multiple of 128 bytes and a multiple of 128 bytes should be transferred, as in other ''raw'' disk interfaces.

Restrictions

Multiple console floppies are not supported.

If a write is given with a count not a multiple of 128 bytes, the trailing portion of the last sector will be zeroed.

Files

`/dev/floppy`

See Also

MAKEDEV(8)

console (4)

Name

console – console interface

Description

Return to the monitor by halting the machine from the operating system or pressing the HALT button on the rear panel. Type `help` at the prompt to receive a description of valid console commands. For example:

```
>>help
```

The `/dev/console` device is valid for all operating system accesses to the character tty device.

Files

`/dev/console`

See Also

tty(4), MAKEDEV(8)

Name

console – console interface

Description

On all but the busless small VAX processors, the console is available to the processor through the console registers. It acts like a normal terminal, except that, when the local functions are not disabled, CTRL/P puts the console in local console mode (where the prompt is ''>>>''). This is true for all processors except the MicroVAX line and busless small VAX processors (see the information that follows). The operation of the console in this mode varies slightly for each processor.

On a VAX-11/780 or VAX-11/785, if you press the BREAK key on the console, the console goes into ODT (console debugger mode). Press P (uppercase letter p) to return from this mode.

On a VAX-11/750 or a VAX-11/730 (11/725), the processor is halted whenever the console is not in conversational mode. Press C (uppercase letter c) to return to conversational mode. When in console mode on a VAX- 11/750 that has a remote diagnosis module, a CTRL/D will put you in remote diagnosis mode, where the prompt will be ''RDM>''. The command `ret` will return from remote diagnosis mode to local console mode.

On any MicroVAX, halt the processor by pressing the HALT button on the front panel. Pressing CTRL/P has no effect on either the MicroVAX-I or the MicroVAX-II. Press c (lowercase letter c) on the console terminal to return to the running system after the HALT button is released.

On the busless small VAX processor, halt the processor by pressing the HALT button on the rear panel (located next to the printer port connector). If the diagnostic console terminal is attached, halt the processor by pressing the BREAK key. For further information, see `ss`(4). Typing CTRL/P has no effect on the busless small VAX processor. Press c (lowercase letter c) on the console terminal to return to the running system.

On an VAX 8800 (8700, 8500, 8550), you can return to conversational mode using the command `sett` (set terminal program) if the processor is still running, or `continue` if it is halted. The `continue` command can be abbreviated to `c`. Halt the processor by pressing CTRL/P. Restart the processor by pressing c (lowercase letter c).

On a VAX 8600 (8650), halt the processor by pressing CTRL/P when the terminal control switch on the front panel is in the LOCAL or REMOTE position. When this switch is in the LOCAL position, pressing CTRL/P at the console terminal is the only way to halt the processor. When the switch is in the REMOTE position, both the console terminal and the remote terminal (if present) can halt the processor. When the terminal control switch is in either LOCAL position, only the console terminal will operate as any other ULTRIX terminal. When the terminal control switch is in either REMOTE position, both the console terminal and the remote terminal will operate as any other ULTRIX terminal.

On a VAX 8100 (8200, 8300), halt the processor by pressing CTRL/P when the console is ENABLED. Press c (lowercase letter c) to return to the running system.

Restrictions

On all but the busless small VAX processors, the console serial line should be used at a relatively low baud rate (2400 BPS or less) as it generally interrupts the CPU for each character received or transmitted. Higher baud rates are permitted but generally impede system performance so much as to warrant using a lower baud rate. On the small VAX, the console operates at 9600 BPS for a terminal or 4800 BPS for a graphics device. For further information, see `ss(4)`.

Files

`/dev/console`

See Also

cty(4), ss(4), tty(4), MAKEDEV(8)

Name

crl – RL02 console interface

Description

This is a simple interface to the RL02 disk unit, which is part of the console subsystem for the VAX 8600 (8650). Access is given to the entire RL02 consisting of 512 cylinders of two tracks of 20 sectors of 256 bytes. The RL02 sectors are accessed as logical 512-byte disk blocks.

All I/O is raw; the seek addresses in raw transfers should be a multiple of 512 and a multiple of 512 bytes should be transferred, as in other ''raw'' disk interfaces.

Restrictions

Only one ''open'' is allowed to the console RL02 device at any given time.

If a write is given with a count not a multiple of 512 bytes, the trailing portion of the last logical block will be zeroed.

The primary purpose of this driver is to apply updates to the console system disk. A ''block'' interface is not provided.

Diagnostics

crl: hard error sn%d crlcs=0x%b, crlds=0x%b
The console subsystem has reported a hard error while performing the requested I/O. The `crlcs` contains standard RLV211 control and status information and `clrds` contains standard drive status information. Bit expansion in ASCII is also provided.

crl: hndshk error
An error in communications between the console subsystem software and the ULTRIX operating system has occurred.

Files

`/dev/crl`

See Also

MAKEDEV(8)

Name

cs – RX50 console interface

Description

This is a simple interface to the RX50 disk unit, which is part of the console subsystem for the VAX 8200 (8200, 8300, 8500, 8550, 8700, 8800). Access is given to the entire RX50 consisting of 80 cylinders of one track of 10 sectors of 512 bytes. The RX50 sectors are accessed as logical 512-byte disk blocks.

The seek addresses in raw transfers should be a multiple of 512 and a multiple of 512 bytes should be transferred, as in other ''raw'' disk interfaces.

Files

```
/dev/cs??
/dev/rcs??
```

See Also

MAKEDEV(8)

Name

ctu – TU58 console interface

Syntax

options MRSP (for VAX-11/750's with an MRSP prom)

Description

Prior to Version 2.0, this device was referenced using `tu(4)`.

The `ctu` interface provides access to the VAX-11/730 (11/725) and VAX-11/750 TU58 console cassette drive.

The interface supports only block I/O to the TU58 cassettes. The devices are normally manipulated with the `arff`(8v) program, using the `f` and `m` options.

The device driver is automatically included when a system is configured to run on a VAX-11/730 (11/725) or VAX-11/750.

The TU58 on a VAX-11/750 uses the Radial Serial Protocol (RSP) to communicate with the CPU over a serial line. This protocol is inherently unreliable as it has no flow control measures built in. On a VAX-11/730 (11/725), the Modified Radial Serial Protocol is used. This protocol incorporates flow control measures that ensure reliable data transfer between the CPU and the device. Certain VAX-11/750s have been modified to use the MRSP prom used in the VAX-11/730 (11/725). To reliably use the console TU58 on an VAX-11/750 under ULTRIX, the MRSP prom is required. For those VAX-11/750s without an MRSP prom, an unreliable but often usable interface has been developed. This interface uses an assembly language ''pseudo-dma'' routine to minimize the receiver interrupt service latency. To include this code in the system, the configuration must not specify the system will run on a VAX-11/730 (11/725) or use an MRSP prom. This unfortunately makes it impossible to configure a single system that will properly handle TU58s on both a VAX-11/750 and an VAX-11/730 (11/725) (unless both machines have MRSP proms).

Restrictions

Frequent data overruns can occur if the VAX-11/750 TU58 is used while in multiuser mode. The interface continues to function and errors are handled, but transfer times may be lengthened considerably.

Diagnostics

tu%d: no bp, active %d
A transmission complete interrupt was received with no outstanding I/O request. This indicates a hardware problem.

tu%d protocol error, state=%s, op=%x, cnt=%d, block=%d
The driver entered an illegal state. The information printed indicates the illegal state, operation currently being executed, the I/O count, and the block number on the cassette.

tu%d receive state error, state=%s, byte=%x
The driver entered an illegal state in the receiver finite state machine. The state is shown along with the control byte of the received packet.

tu%d: read stalled
A timer watching the controller detected no interrupt for an extended period while an operation was outstanding. This usually indicates that one or more receiver interrupts were lost and the transfer is restarted (VAX-11/750 only).

tu%d: hard error bn%d, pk_mod %o
The device returned a status code indicating a hard error. The actual error code is shown in octal. No retries are attempted by the driver.

Files

`/dev/tu0`	
`/dev/tu1`	(only on a VAX-11/730 (11/725))

See Also

MAKEDEV(8)

Name

cty – extra serial line interface

Description

The extra serial lines are available to the processor through the console registers. These lines are available on the VAX 8100, 8200, 8300, 8500, 8550, 8600, 8650, 8700, and 8800 machines and number three all total. They act like normal terminal lines, except for the VAX 8600 (8650), where the first line can also act as a remote console port and the other two lines are currently unused. Entries for these lines should be placed in `/etc/ttys`, if they are used.

There are also certain workstation configurations that may use the console serial interface. For MicroVAX workstation configurations, calling `MAKEDEV`(8) with the argument ttycp will create the device special file needed to utilize the console interface. Before using the serial line interface, it is important to consider the setting of the halt enable switch on a MicroVAX workstation.

Restrictions

In general, the lines are low-performance devices. They can be used as terminal ports, but, for each character sent or received, the CPU is interrupted. Thus, it is recommended that the extra serial lines be used at low baud rates (2400 baud or less), so that console input and output do not severely impact system performance.

Files

`/dev/ttyc?`

See Also

console(4), tty(4), MAKEDEV(8)

Name

cxa – CXA16 communications interface

Syntax

device dhu0 at uba0 csr 0160440 flags 0xffff vector dhurint dhuxint

Description

A CXA16 provides 16 data-leads-only communication lines with no modem control. The CXA16 conforms to RS423A. The device behaves and looks just like a DHV11 (with the exception of modem control and number of lines) and is specified in the configuration line the same as a DHV11 device. Each line attached to the CXA16 communications multiplexer behaves as described in `tty`(4) and can be set to run at any of 16 speeds; see `tty`(4) for the encoding.

A flags field of 0xffff must be used to specify that all lines are to operate as hardwired. This is done to prevent the line from being treated as a modem control line.

The dhu driver normally interrupts on each input character.

NOTE

The `cxa`, `cxb`, `cxy`, `dhv`, and `dhq` devices operate under the control of the `dhu` driver.

Diagnostics

dhu%d: receive fifo overflow
The character input fifo overflowed before it could be serviced. This can happen if a hard error occurs when the CPU is running with elevated priority, as the system then prints a message on the console with interrupts disabled. This can cause a few input characters to be lost to users. It is not serious.

dhu%d:%d DMA ERROR
A DMA transfer error has occurred. The `cxa` unit number and line number are printed. This means that the channel indicated has failed to transfer DMA data within 10.7 microseconds of the bus request being acknowledged or that there is a memory parity error. This may cause a few output characters to be lost.

dhu%d: DIAG. FAILURE
This indicates that the `cxa` internal diagnostics have detected an error.

dhu%d: DHU HARDWARE ERROR. TX.DMA.START failed
The `cxa` failed to clear the start bit. Normally, this is cleared to signal that a DMA transfer has completed.

Files

`/dev/tty??`

See Also

tty(4), MAKEDEV(8)

cxb(4)

Name

cxb – CXB16 communications interface

Syntax

device dhu0 at uba0 csr 0160440 flags 0xffff vector dhurint dhuxint

Description

A CXB16 provides 16 data leads-only communication lines with no modem control. The CXB16 conforms to RS422A. The device behaves and looks just like a DHV11 (with the exception of modem control and number of lines) and is specified in the configuration line the same as a DHV11 device. Each line attached to the CXB16 communications interface behaves as described in `tty`(4) and can be set to run at any of 16 speeds. See `tty`(4) for the encoding.

A flags field of 0xffff must be used to specify that all lines are to operate as hardwired. This is done to prevent the line from being treated as a modem control line.

The dhu driver normally interrupts on each input character.

NOTE

The `cxa`, `cxb`, `cxy`, `dhv`, and `dhq` devices operate under the control of the dhu driver.

Diagnostics

dhu%d: receive fifo overflow
The character input fifo overflowed before it could be serviced. This can happen if a hard error occurs when the CPU is running with elevated priority, as the system then prints a message on the console with interrupts disabled. This can cause a few input characters to be lost to users. It is not serious.

dhu%d:%d DMA ERROR
A DMA transfer error has occurred. The `cxb` unit number and line number are printed. This means that the channel indicated has failed to transfer DMA data within 10.7 microseconds of the bus request being acknowledged or that there is a memory parity error. This may cause a few output characters to be lost.

dhu%d: DIAG. FAILURE
This indicates that the `cxb` internal diagnostics have detected an error.

dhu%d: DHU HARDWARE ERROR. TX.DMA.START failed
The `cxb` failed to clear the start bit. Normally, this is cleared to signal that a DMA transfer has completed.

Files

`/dev/tty??`

See Also

tty(4), MAKEDEV(8)

Name

cxy – CXY08 communications interface

Syntax

device dhu0 at uba0 csr 0160440 flags 0x?? vector dhurint dhuxint

Description

A CXY08 provides eight communication lines with modem control adequate for UNIX dialup use. The device behaves and looks just like a DHV11 and is specified in the configuration line the same as a DHV11 device. Each line attached to the CXY08 communications interface behaves as described in `tty`(4) and can be set to run at any of 16 speeds. See `tty`(4) for the encoding.

Bit *i* of flags can be specified for a `cxy` device to say that a line is not properly connected and that the line should be treated as hardwired with carrier always present. Thus, specifying flags 0x04 in the specification of `dhu0` would cause line tty02 to be treated in this way.

The `dhu` driver normally interrupts on each input character.

NOTE

The `cxa`, `cxb`, `cxy`, `dhv`, and `dhq` devices operate under the control of the `dhu` driver.

Diagnostics

dhu%d: receive fifo overflow
The character input fifo overflowed before it could be serviced. This can happen if a hard error occurs when the CPU is running with elevated priority, as the system then prints a message on the console with interrupts disabled. This can cause a few input characters to be lost to users. It is not serious.

dhu%d:%d DMA ERROR
A DMA transfer error has occurred. The `cxy` unit number and line number are printed. This means that the channel indicated has failed to transfer DMA data within 10.7 microseconds of the bus request being acknowledged or that there is a memory parity error. This may cause a few output characters to be lost.

dhu%d: DIAG. FAILURE
This indicates that the `cxy` internal diagnostics have detected an error.

dhu%d: DHU HARDWARE ERROR. TX.DMA.START failed
The `cxy` failed to clear the start bit. Normally, this is cleared to signal that a DMA transfer has completed.

Files

`/dev/tty??`	
`/dev/ttyd?`	Dialups

See Also

tty(4), MAKEDEV(8)

Name

dc – serial line/mouse/keyboard

Syntax

device dc0 at ibus? vector dcintr

Description

The serial line controller provides four ports, with modem control on two of the ports. The DECstation 3100 and DECstation 2100 only provide partial modem control. The DECstation 5000 provides full modem control. The ports are used as follows:

Port	**Usage**
0	Graphics device keyboard at 4800 BPS
1	Mouse or tablet at 4800 BPS
2	Communications port 1 (w/modem control)/local terminal
3	Communications port 2 (w/modem control)/local terminal

Each communication port from the serial line controller behaves as described in `tty`(4) and can be set to run at any of 16 speeds. For the encoding, see `tty`(4).

When a graphics device is not being used as the system console, communications port 2 becomes the system console. In this configuration, the port can only be used at 9600 BPS and no modem control is supported.

The serial line driver operates in interrupt-per-character mode (all pending characters are flushed from the silo on each interrupt).

Restrictions

Speed must be set to 9600 BPS on the console port and 4800 BPS on ports used by graphics devices. The serial line driver enforces this restriction; that is, changing speeds with the `stty` command may not always work on these ports.

Files

`/dev/console`	console terminal
`/dev/tty00`	local terminal
`/dev/tty01`	local terminal

See Also

console(4), devio(4), tty(4), ttys(5), MAKEDEV(8)

Name

de – DEUNA/DELUA Ethernet interface

Syntax

device de0 at uba0 csr 0174510 vector deintr

Description

The `de` interface provides access to a 10 Mb/s Ethernet network through a DEUNA/DELUA controller.

The host's Internet address is specified at boot time with an SIOCSIFADDR ioctl. The `de` interface employs the address resolution protocol described in `arp`(4p) to dynamically map between Internet and Ethernet addresses on the local network.

The interface normally tries to use a ''trailer'' encapsulation to minimize copying data on input and output. This can be disabled for an interface by setting the IFF_NOTRAILERS flag with an SIOCSIFFLAGS ioctl. Trailers are only used for packets destined for Internet hosts.

The SIOCSPHYSADDR ioctl can be used to change and SIOCRPHYSADDR can be used to read the physical address of the board.

SIOCADDMULTI and SIOCDELMULTI can be used to add or delete multicast addresses. The board recognizes, at most, 10 multicast addresses. The argument to these ioctls is a pointer to an ifreq structure found in <`net/if.h`>.

SIOCRDCTRS and SIOCRDZCTRS ioctls can be used to read or ''read and clear'' the board counters. The argument to these two ioctls is a pointer to a counter structure ''ctrreq'' found in <`net/if.h`>.

The ioctls SIOCENABLBACK and SIOCDISABLBACK can be used to enable and disable the interface loopback mode.

Restrictions

The PUP protocol family is not supported.

Diagnostics

de%d: command failed, csr0=%b csr1=%b
Here, **command** is one of **reset, pcbb, rdphyad, wtring,** or **wtmode**. This message is printed if there is an error on device initialization. The following command failures can occur during ioctl requests:

wtphyadd
An attempt to change the physical address failed.

rdphyadd
An attempt to read the physical address failed.

wtmulti
An attempt to add a new multicast address failed.

mtmulti failed, multicast list full
An attempt to add a new multicast address failed because the maximum number of multicast addresses has been reached.

rdcnts
An attempt to read the board counters failed.

The following messages occur while transmitting or receiving packets:

de%d: buffer unavailable
Packets are being received by the interface faster than they can be serviced by the driver.

de%d: can't handle af%d
The interface was handed a message with addresses formatted in an unsuitable address family; the packet was dropped.

See Also

arp(4p), inet(4f), intro(4n)

Name

devio – device information interface

Syntax

#include <sys/ioctl.h>
#include <sys/devio.h>

Description

The `devio` interface obtains status, device attributes, and other information by polling the controlling device driver. There are two ioctl requests associated with this interface: DEVIOCGET and DEVGETGEOM.

The file `<sys/devio.h>` contains the needed structure and definitions to use the DEVIOCGET and DEVGETGEOM ioctl requests.

The DEVIOCGET ioctl request is used to obtain generic device information by polling the underlying device driver. The following example shows the basic structure used by this request:

```
/* Structure for DEVIOCGET ioctl - device get status command */

struct devget  {
   short    category;            /* Category                    */
   short    bus;                 /* Bus                         */
   char     interface[DEV_SIZE]; /* Interface (string)          */
   char     device[DEV_SIZE];    /* Device (string)             */
   short    adpt_num;            /* Adapter number              */
   short    nexus_num;           /* Nexus or node on adapter #  */
   short    bus_num;             /* Bus number                  */
   short    ctlr_num;            /* Controller number           */
   short    slave_num;           /* Plug or line number         */
   char     dev_name[DEV_SIZE];  /* ULTRIX device mnemonic      */
   short    unit_num;            /* ULTRIX device unit number   */
   unsigned soft_count;          /* Driver soft error count     */
   unsigned hard_count;          /* Driver hard error count     */
   long     stat;                /* Generic status mask         */
   long     category_stat;       /* Category specific mask      */
};
```

The DEVGETGEOM ioctl request is used to obtain disk geometry and attributes by polling the underlying device driver. This ioctl request is only supported on MSCP and SCSI disk drivers. The ioctl fails on other types of drivers which do not support DEVGETGEOM. The ioctl may fail if the device driver is unable to obtain geometry information. This could happen if the disk media is removable and there is no media loaded in the drive.

The following example shows the basic structure used by this request:

```
/* Structure for DEVGETGEOM ioctl - disk geometry information */

typedef union devgeom {
  struct {
     unsigned long  dev_size;    /* number of blocks in user area   */
     unsigned short  ntracks;    /* number of tracks per cylinder   */
     unsigned short  nsectors;   /* number of sectors per track     */
     unsigned short  ncylinders; /* total number of cylinders       */
     unsigned long   attributes; /* Device attributes               */
   } geom_info;
   unsigned char       pad[124];  /* Allocate space for expansion */
} DEVGEOMST;
```

The following is a description of the fields of the DEVGEOMST data structure. Many of the fields correspond to attributes that are often specified in the disk description file `/etc/disktab`. This ioctl is used by the `creatdiskbyname` subroutine to dynamically generate disktab entries.

dev_size	This field contains the number of user accessable blocks on the disk. The corresponding disktab field is `pc`, which describes the size of the "c" partition.
ntracks	This field contains the number of tracks per cylinder and corresponds to the `nt` field of a disktab entry.
nsectors	This field contains the number of sectors per track and corresponds to the `ns` field of a disktab entry.
ncylinders	This field contains the number of cylinders on the disk and corresponds to the `nc` field of a disktab entry.
attributes	This field represents disk attributes.
pad	This field is not used to store disk information. The pad element of the DEVGEOMST is used to provide room for future expansion of the information fields.

Restrictions

The DEVGETGEOM ioctl request is only supported on MSCP and SCSI disk drivers.

See Also

creatediskbyname(3x), ra(4), rz(4), disktab(5)

Name

dfa – DFA01 communications interface

Syntax

device dz0 at uba0 csr 0160100 flags 0x0 vector dzrint dzxint

Description

The DFA01 contains two DF224-compatible modems with a DZQ11-compatible interface. Each line attached to the DFA01 behaves as described in `tty`(4). See `tty`(4) for the encoding. Lines may operate at 300, 1200, or 2400 baud.

Caution

The DZQ11 interface is capable of baud rates up to 9600 baud, but, because the modem is restricted to speeds of 300, 1200, and 2400 baud, all other baud rates are considered illegal and pass meaningless data.

A flags field of 0x0 must be specified to indicate that the lines are to be treated as modems.

An `acucap`(5) entry of `dfa01` should be used to describe the 2400 baud autodial attributes used by such programs as `tip`(1c), `cu` and `uucp`(1c). For pulse tone dialing at 2400 baud, an `acucap`(5) entry of `dfa01-p` should be used. To dial at 1200 baud, the `dfa01-1200` or `dfa01-1200p` (pulse dial) should be used. To use the DFA01 autodialer, the terminal line must be set to no parity.

Any entry in the `remote`(5) file that specifies an ACU type of `dfa01` must specify no parity. The following example shows how a correct entry in the `remote`(5) file may look:

```
dial2400|2400 Baud attributes:\
        :dv=/dev/ttyd0:br#2400:at=dfa01:du:pa=none:
```

The `dfa` driver normally uses its input silos and polls for input at each clock tick (10 milliseconds) rather than taking an interrupt on each input character.

`MAKEDEV`(8) will produce four terminal lines for the DZQ11 interface. The second and fourth lines are the actual modem lines. The first and third lines pass status information when the modem is on line.

Diagnostics

dz%d: receive fifo overflow.
The character input fifo overflowed before it could be serviced. This can happen if other devices heavily utilize the bus and CPU, preventing interrupts from the DFA01 from being serviced. This may cause a few input characters to be lost to users. It is not serious.

Files

`/dev/tty??`	
`/dev/ttyd?`	Dialups

See Also

tty(4), acucap(5), MAKEDEV(8)

Name

dhb – DHB32 communications multiplexer

Syntax

device dmb0 at vaxbi? node? flags 0x????
vector dmbsint dmbaint dmblint

Description

A DHB32 device provides 16 asynchronous communication lines with full modem control. The DHB32 and the DMB32 share a common software device driver. For this reason, the configuration line is the same for both the DHB32 and DMB32.

Each line attached to a DHB32 serial line port behaves as described in `tty`(4). Input and output for each line can independently be set to run at any of 16 speeds. See `tty`(4) for the encoding.

Bit *i* of flags may be specified for a `dhb` to say that a line should be treated as a hardwired connection with carrier always present. If bit *i* of flags is not set, the line will operate under full modem control. Modem lines will operate in accordance to the CD (carrier detect), DSR (data set ready) and CTS (clear to send) leads. Thus, specifying ''flags 0x0004'' in the specification of `dmb0` would cause line 2 on the DHB32 to be treated as hardwired with carrier always present. In this example, the remainder of the lines will be modem control lines.

Diagnostics

dmbinit: async lines unavailable
This message is produced at system boot time if the DHB32 fails its internal self test indicating that the asynchronous lines have failed to configure.

dmb%d: fifo overflow
The character input fifo overflowed before it could be serviced. This can happen if the CPU is running with elevated priority for too long a period of time. Overflow errors may indicate that configuration constraints have been reached.

dmb%d: DMA Error. tbuf = 0x%x
A DMA output transfer failed. This problem can result from a memory error or an invalid pte (page table entry). For a description of the error code in the ''tbuf'' register. See the DHB32 documentation.

dmb%d: DMB Hardware Error. TX.DMA.START failed
The `dhb` failed to clear the start bit. Normally, this is cleared to signal that a DMA transfer has completed.

dmb%d: Modem Error. tbuf = 0x%x
Indicates a problem with a modem or its cable. For a description of the error code in the ''tbuf'' register. See the DHB32 documentation.

dmb%d: Internal Error. tbuf = 0x%x
Indicates that the DHB32 detected an internal error. For a description of the error code in the ''tbuf'' register. See the *DHB32 User Guide.*

Files

`/dev/tty??`	
`/dev/ttyd?`	(modem lines only)

See Also

tty(4), MAKEDEV(8), dmbsp(4)

Name

dhq – DHQ11 communications interface

Syntax

device dhu0 at uba? csr 0160440 flags 0x?? vector dhurint dhuxint

Description

A DHQ11 provides eight communication lines with modem control.

Each line attached to the DHQ11 communications multiplexer behaves as described in `tty`(4). Input and output for each line can be set independently to run at any of 16 speeds. See `tty`(4) for the coding.

Bit *i* of flags can be specified for a DHQ11 to say that a line is not properly connected and that the line should be treated as hardwired with carrier always present. Thus, specifying ''flags 0x04'' in the specification of `dhu0` would cause the third line to be treated in this way.

NOTE

The `dhq` driver operates under the control of the `dhu` driver.

Diagnostics

dhu%d %d: DMA ERROR
The indicated channel failed to transfer DMA data within 21.3 microseconds of the bus request being acknowledged or there was a memory parity error.

dhu%d: DIAG. FAILURE
The DHQ11 failed the diagnostics that run at initialization time.

dhu%d: recv. fifo overflow
The character input fifo overflowed before it could be serviced. This can happen if a hard error occurs when the CPU is running with elevated priority. Interrupts are disabled, and the system then prints a message on the console.

dhu%d: DHU HARDWARE ERROR. TX.DMA.START failed
The dhq failed to clear the start bit. Normally, this is cleared to signal that a DMA transfer has completed.

Files

`/dev/tty??`	
`/dev/ttyd?`	Dialups

See Also

tty(4), MAKEDEV(8)

Name

dhu – DHU11 communications interface

Syntax

device dhu0 at uba? csr 0160440 flags 0x???? vector dhurint dhuxint

Description

A DHU11 provides 16 communication lines with modem control.

Each line attached to the DHU11 communications multiplexer behaves as described in `tty`(4). Input and output for each line can be set independently to run at any of 16 speeds. See `tty`(4) for the coding.

Bit *i* of flags can be specified for a DHU11 to say that a line is not properly connected and that the line should be treated as hard-wired with carrier always present. Thus, specifying ''flags 0x0004'' in the specification of `dhu0` would cause the third line tty02 to be treated in this way.

Diagnostics

dhu%d %d: DMA ERROR
The indicated channel failed to transfer DMA data within 21.3 microseconds of the bus request being acknowledged or there was a memory parity error. This is often followed by a UNIBUS adapter error, which occurs most frequently when the UNIBUS is heavily loaded and when devices such as rk07s, which monopolize the bus, are present.

dhu%d: DHU HARDWARE ERROR. TX.DMA.START failed
The `dhu` failed to clear the start bit. Normally, this is cleared to signal that a DMA transfer has completed.

dhu%d: DIAG. FAILURE
The DHU11 failed the diagnostics that run at initialization time.

dhu%d: recv. fifo overflow The character input fifo overflowed before it could be serviced. This can happen if a hard error occurs when the cpu is running with elevated priority. Interrupts are disabled, and the system then prints a message on the console.

Files

`/dev/tty??`

`/dev/ttyd?`   (modem lines only)

See Also

tty(4), MAKEDEV(8)

Name

dhv – DHV11 communications interface

Syntax

device dhu0 at uba0 csr 0160440 flags 0x?? vector dhurint dhuxint

Description

A DHV11 provides eight communication lines with partial modem control, adequate for UNIX dialup use. Each line attached to the DHV11 communications interface behaves as described in `tty`(4) and can be set to run at any of 16 speeds. See `tty`(4) for the encoding.

Bit *i* of flags may be specified for a DHV to say that a line is not properly connected and that the line should be treated as hardwired, with carrier always present. Thus, specifying ''flags 0x04'' in the specification of dhu0 would cause line tty02 to be treated in this way.

The `dhv` driver normally interrupts on each input character.

NOTE

The `cxa`, `cxb`, `cxy`, `dhv`, and `dhq` devices operate under the control of the `dhu` driver.

Diagnostics

dhu%d: receive fifo overflow
The character input fifo overflowed before it could be serviced. This can happen if a hard error occurs when the CPU is running with elevated priority, as the system will then print a message on the console with interrupts disabled. This can cause a few input characters to be lost to users. It is not serious.

dhu%d:%d DMA ERROR
A DMA transfer error has occurred. The `dhv` unit number and line number are printed. This means that the channel indicated has failed to transfer DMA data within 10.7 microseconds of the bus request being acknowledged or that there is a memory parity error. This may cause a few output characters to be lost.

dhu%d: DIAG. FAILURE
This indicates that the `dhv` internal diagnostics have detected an error.

dhu%d: DHU HARDWARE ERROR. TX.DMA.START failed
The `dhv` failed to clear the start bit. Normally, this is cleared to signal that a DMA transfer has completed.

Files

`/dev/tty??`	
`/dev/ttyd?`	Dialups

See Also

tty(4), MAKEDEV(8)

Name

dkio – disk interface

Syntax

#include <sys/fs.h>
#include <sys/ioctl.h>

Description

This section describes the ioctl (input/output controller) codes for all disk drivers. The basic ioctl (input/output controller) format is:

#include <sys/fs.h>
#include <sys/ioctl.h>
ioctl(*fildes, code, arg)*
struct pt **arg*;

The applicable *codes* are:

DIOCGETPT	Indicates to the driver to store the information in the current partition table in the address pointed to by *arg*. The file descriptor must be opened on the raw partitions, *a* or *c*. DIOCGETPT does not change the partition table, but it does provide access to the partition table information.
DIOCSETPT	Indicates to the driver to modify the current partition table with the information pointed to by *arg*. The file descriptor must be opened on the raw partitions, *a* or *c*. If the *a* or *c* partition is not mounted, only the partition table in the driver is modified. This temporarily modifies the partition table of the disk. The modifications are overwritten with the default table when the disk is turned off and on. If the *a* or *c* partition is mounted, both the partition table in the driver and the partition table in the primary superblock are modified. This permanently modifies the partition table of the disk. This is not recommended. To change a partition table permanently, use the `chpt`(8) command.
DIOCDGTPT	Indicates to the driver to store the *default* information of the current partition table in the address pointed to by *arg*. The file descriptor must be opened on the raw partitions *a* or *c*. DIOCGETPT does not change the partition table, but it does provide access to the partition table information.
DKIOCGET	Allows the user to receive generic disk information as defined in `<sys/devio.h>` *struct*devget.
DKIOCACC	This code is defined in `<sys/bbr.h>`. It is currently unused.

Restrictions

These restrictions apply when using the DIOCSETPT ioctl code:

- You must have superuser privileges.
- You cannot shrink or change the offset of a partition with a file system mounted on it or with an open file descriptor on the entire partition.
- You cannot change the offset of the *a* partition.

Examples

This example shows how to use the DIOGETPT ioctl code to print the length and offset of the *a* partition of an RZ23 disk:

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/fs.h>
#include <sys/ioctl.h>

main()
{
        struct pt arg;
        int fd, i;

        /* Open the "a" partition of the disk you want to see */

        if ( (fd = open("/dev/rz0a",0)) < 0 ) {
                printf("Unable to open device\n");
                exit(2);
        }

        /* Get the partition information */

        if ( ioctl(fd,DIOCGETPT,&arg) < 0 )
                printf("Error in ioctl\n");

        printf("Length\t\tOffset\n");


        for ( i = 0; i <= 7; i++ ) {
                printf("%d\t\t%d\n",arg.pt_part[i].pi_nblocks,
                                   arg.pt_part[i].pi_blkoff );
        }
}
```

Files

`/dev/{r}rz???`

See Also

rz(4), disktab(5), fstab(5), chpt(8), diskpart(8), fsck(8), MAKEDEV(8), mkfs(8), tunefs(8)

Name

dkio – disk interface

Syntax

#include <sys/fs.h>
#include <sys/ioctl.h>

Description

This section describes the ioctl (input/output controller) codes for all disk drivers. The basic ioctl (input/output controller) format is:

#include <sys/fs.h>
#include <sys/ioctl.h>
ioctl(*fildes, code, arg*)
struct pt **arg*;

The applicable *codes* are:

DIOCGETPT Indicates to the driver to store the information in the current partition table in the address pointed to by *arg*. The file descriptor must be opened on the raw partitions, *a* or *c*.

DIOCGETPT does not change the partition table, but it does provide access to the partition table information.

DIOCSETPT Indicates to the driver to modify the current partition table with the information pointed to by *arg*.

The file descriptor must be opened on the raw partitions, *a* or *c*.

If the *a* or *c* partition is not mounted, only the partition table in the driver is modified. This temporarily modifies the partition table of the disk. The modifications are overwritten with the default table when the disk is turned off and on.

If the *a* or *c* partition is mounted, both the partition table in the driver and the partition table in the primary superblock are modified. This permanently modifies the partition table of the disk. This is not recommended. To change a partition table permanently, use the `chpt`(8) command.

DIOCDGTPT Indicates to the driver to store the *default* information of the current partition table in the address pointed to by *arg*. The file descriptor must be opened on the raw partitions, *a* or *c*.

DIOCGETPT does not change the partition table, but it does provide access to the partition table information.

DKIOCGET Allows the user to receive generic disk information as defined in `<sys/devio.h>` *struct devget.*

DKIOCACC This code is defined in `<sys/bbr.h>`. For an MSCP class disk, the driver performs one of the following functions:

ACC_REVEC

Forces revector of a specified disk block.

ACC_SCAN
Scans an area of the disk reporting any forced errors found and revectoring any bad blocks found.

DKIOCEXCL This command is used to set and clear the exclusive access attribute on controllers that provide multihost support. In this case *arg* is an integer pointer. If the value of *arg* is 0 the exclusive access attribute will be cleared. If the value of *arg* is nonzero the exclusive access attribute will be set. The exclusive access attribute is set on a per-drive basis and can not be used selectively on individual partitions.

Attempts to clear the exclusive attribute will fail if the drive is not currently set exclusive access to the issuing host or the underlying controller or driver does not support multihost exclusive access. Attempts to set the exclusive attribute will fail if the drive is already exclusively associated with another host or the underlying controller or driver does not support multihost exclusive access.

Examples

This example shows how to use the DIOGETPT ioctl code to print the length and offset of the *a* partition of an RA81 disk:

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/fs.h>
#include <sys/ioctl.h>

main()
{
       struct pt arg;
       int fd, i;

       /* Open the "a" partition of the disk you want to see */

       if ( (fd = open("/dev/rra0a",0)) < 0 ) {
             printf("Unable to open device\n");
             exit(2);
       }

       /* Get the partition information */

       if ( ioctl(fd,DIOCGETPT,&arg) < 0 )
             printf("Error in ioctl\n");

       printf("Length\t\tOffset\n");


       for ( i = 0; i <= 7; i++ ) {
             printf("%d\t\t%d\n",arg.pt_part[i].pi_nblocks,
                           arg.pt_part[i].pi_blkoff );
       }
}
```

Restrictions

These restrictions apply when using the DIOCSETPT ioctl code:

- You must have superuser privileges.
- You cannot shrink or change the offset of a partition with a file system mounted on it or with an open file descriptor on the entire partition.
- You cannot change the offset of the *a* partition.
- The exclusive access ioctl DKIOCEXCL is only supported on HSC version V5.00 or later.

Files

```
/dev/{r}ra???
/dev/{r}hp???
/dev/{r}rb???
/dev/{r}rd???
/dev/{r}rk???
/dev/{r}rl???
/dev/{r}rx???
```

See Also

hp(4), ra(4), rb(4), rd(4), rk(4), rl(4), rx(4), disktab(5), fstab(5), chpt(8), diskpart(8), fsck(8), MAKEDEV(8), mkfs(8), tunefs(8)

dmb(4)

Name

dmb – DMB32 communications multiplexor

Syntax

device dmb0 at vaxbi? node? flags 0x????
        vector dmbsint dmbaint dmblint

Description

A DMB32 device provides eight asynchronous communication lines with modem control. The device driver also supports a connection to a line printer through the line printer port of the DMB32.

Each line attached to a DMB32 serial line port behaves as described in `tty`(4). Input and output for each line may independently be set to run at any of 16 speeds. See `tty`(4) for the encoding.

Bit *i* of flags can be specified for a `dmb` to say that a line should be treated as a hardwired connection, with carrier always present. If bit *i* of flags is not set, the line operates under full modem control. Modem lines will operate in accordance to the CD (carrier detect), DSR (data set ready) and CTS (clear to send) leads. Thus, specifying ''flags 0x0004'' in the specification of `dmb0` would cause line 2 on the DMB32 to be treated as hardwired, with carrier always present. In this example, the remainder of the lines will be modem control lines.

Restrictions

The DMB32 provides a synchronous port, but this is not supported by the driver.

Diagnostics

dmbinit: async lines unavailable
This message is produced at system boot time, if the DMB32 fails its internal self test, indicating that the asynchronous lines have failed to configure.

dmbinit: printer port unavailable
This message is produced at system boot time, if the DMB32 fails its internal self test, indicating that the printer port failed to configure.

dmb%d: fifo overflow
The character input fifo overflowed before it could be serviced. This can happen if the CPU is running with elevated priority for too long a period of time. Overflow errors can indicate that configuration constraints have been reached.

dmb%d: DMA Error. tbuf = 0x%x
A DMA output transfer failed. This can be caused by a memory error or an invalid pte (page table entry). For a description of the error code in the ''tbuf'' register, see the DMB32 documentation.

dmb%d: DMB Hardware Error. TX.DMA.START failed
The `dmb` failed to clear the start bit. Normally, this is cleared to signal that a DMA transfer has completed.

dmb%d: Modem Error. tbuf = 0x%x
Indicates a problem with a modem or its cable. For a description of the error code in the ''tbuf'' register, see the DMB32 documentation.

dmb%d: Internal Error. tbuf = 0x%x
Indicates that the DMB32 detected an internal error. For a description of the error code in the ''tbuf'' register, see the DMB32 documentation.

hfBdmb%d: uio move error
An error occurred when copying a printer buffer from user space to system space.

dmbsint
The unsupported synchronous port of the DMB32 interrupted because of a problem. Check your interrupt vectors for a conflict with another device.

Files

`/dev/tty??`

`/dev/ttyd?` (modem lines only)

See Also

tty(4), MAKEDEV(8), dmbsp(4)

dmbsp(4)

Name

dmbsp – DMB32 serial printer interface

Syntax

device dmb0 at vaxbi? node? flags 0x????
vector dmbsint dmbaint dmblint

Description

A DMB32 supports a connection to a line printer through the line printer port of the DMB32.

Bits 8-15 of the *flags* longword specify the number of columns per line on the line printer. If 0 is specified, 132 columns are used. Bits 16-23 specify the number of lines per page. If 0 is specified, 66 are used.

Diagnostics

dmb%d: Line Printer Disconnected.
This message occurs if an open is performed on the printer port when the DMB32 detects that there is no printer cable connected.

dmb%d: Printer DMA Error
A DMA output transfer to the printer failed. This can be caused by a memory error or an invalid pte (page table entry).

Files

`/dev/lp?`

See Also

dmb(4), MAKEDEV(8)

Name

dmc – DMC11/DMR11 communications interface

Syntax

device dmc0 at uba0 csr 0167600 flags 0x???? vector dmcrint dmcxint

Description

The dmc interface provides access to a point-to-point communications device that runs at either 1 Mb/s or 56 Kb/s. DMC11s communicate using the DDCMP link layer protocol.

The dmc interface driver also supports a DMR11 providing point-to-point communication running at data rates from 2.4 Kb/s to 1 Mb/s. DMR11s are a more recent design and are preferred over DMC11s.

The host address must be specified with an SIOCSIFADDR ioctl before the interface will transmit or receive any packets.

Several protocols can be multiplexed over a dmc link simultaneously. Conversely, a dmc can be set up such that only one protocol family can use that device. If the latter approach is taken, an SIOCSTATE ioctl must be issued by the protocol family requesting device ownership. The family address must appear in the ''if_family'' structure member, and ''if_nomuxhdr'' must be set. Before requesting ownership, make sure that access to the device for all other protocol families is disabled.

The first byte of the *flags* word can be set up to indicate what mode the device should use. The supported modes are 0 for full duplex, 1 for maintenance mode, and 2 for half duplex. In addition, if the device is a dmr, the number of outstanding transmit buffers can be increased from a default of 7 to a maximum of 24 buffers by specifying a hexadecimal value in the second byte of the *flags* word. For example, if the *flags* word is set to 0x1800, 24 transmit buffers will be allocated on a device set up to run full duplex.

Restrictions

Note that maintenance mode should be used only to diagnose data link problems. It is not intended to be used for normal data link traffic.

Diagnostics

dmcprobe: can't start device
The dmc could not be started at boot time.

dmcinit: DMC not running
The dmc unexpectedly stopped running.

dmc%d: done unalloc rbuf
The dmc returned a receive or transmit buffer that was not allocated to it.

dmc%d: bad control %o
A bad parameter was passed to the *dmcload* routine.

dmc%d: unknown address type %d
An input packet was received that contained a type of address unknown to the driver.

dmc%d: bad packet address 0x%x
The device returned a buffer with an unexpected buffer address.

dmc%d: can't handle af%d
The interface was handed a message that has addresses formatted in an unsuitable address family. Formerly reported as **dmc%d: af%d not supported.**

dmc%d: internal loopback enable requested
The device is being put in internal loopback at a user's request.

dmc%d: internal loopback disable requested
The device is being taken out of internal loopback at a user's request.

DMC FATAL ERROR 0%o

DMC SOFT ERROR 0%o

See Also

inet(4f), intro(4n)

Name

dmf – DMF32 communications interface

Syntax

device dmf0 at uba? csr 0160340 flags 0x????
vector dmfsrint dmfsxint dmfdaint dmfdbint dmfrint dmfxint dmflint

Description

The `dmf` device provides eight lines of asynchronous serial line support with full modem control on two lines only. The device driver also supports a connection to a line printer through the line printer port of the DMF32.

Each line attached to a DMF32 serial line port behaves as described in `tty`(4). Input and output for each line can be set independently to run at any of 16 speeds. See `tty`(4) for the encoding.

Bit *i* of `flags` can be specified for a `dmf` to say that a line is not properly connected, and that the line should be treated as hardwired, with carrier always present. Thus, specifying ''flags 0x00f6'' in the specification of `dmf0` would cause lines 0 and 1 on the DMF32 to be treated as modem lines, while lines 2 through 7 are direct connect no-modem lines. It is important to specify lines 2 through 7 as direct connect, because the device does not support modem control on these lines.

The `dmf` driver normally uses input silos and polls for input at each clock tick (10 milliseconds).

Caution

The DMF32 will discard incoming characters on the lines with full modem control, if carrier is not present.

Restrictions

The DMF32 provides other services, but these are not supported by the driver.

Diagnostics

dmf%d: NXM line %d
No response from UNIBUS on a `dma` transfer within a timeout period. This is often followed by a UNIBUS adapter error. This occurs most frequently when the UNIBUS is heavily loaded and when devices that monopolize the bus, such as RK07s, are present. It is not serious.

dmf%d: silo overflow
The character input silo overflowed before it could be serviced. This can happen if a hard error occurs when the CPU is running with elevated priority, as the system then prints a message on the console with interrupts disabled.

dmfsrint
dmfsxint
dmfdaint
dmfdbint
One of the unsupported ports of the DMF32 interrupted because of a problem. Check your interrupt vectors for a conflict with another device.

Files

```
/dev/tty??
/dev/ttyd?    (modem lines only)
```

See Also

dmfsp(4), tty(4), MAKEDEV(8)

Name

dmfsp – DMF32 serial printer

syntax

device dmf0 at uba? csr 0160340 flags 0x????
    vector dmfsrint dmfsxint dmfdaint dmfdbint dmfrint dmfxint dmflint

Description

The `dmf` supports a connection to a line printer through the line printer port of the DMF32.

Bits *8-15* of the `flags` longword are used to specify the number of columns per line on the line printer. If 0 is specified, 132 columns are used. Bits *16-23* are used to specify the number of lines per page. If 0 is specified, 66 lines are used.

Diagnostics

dmf%d: Line Printer Disconnected
This message occurs if an open is performed on the printer port when the DMF32 detects that there is no printer cable connected.

dmf%d: Printer DMA Error
A DMA output transfer to the printer failed. This could be caused by a memory error or an invalid pte (page table entry).

Files

`/dev/lp?`

See Also

dmf(4), MAKEDEV(8)

Name

dmv – DMV11 communications interface

Syntax

device dmv0 at uba0 csr 0167600 flags 0x???? vector dmvrint dmvxint

Description

The `dmv` interface provides access to point-to-point communications that runs at speeds from 2.4 Kb/s to 56 Kb/s. DMV11s communicate using the DDCMP link layer protocol.

Several protocols can be multiplexed over a `dmv` link simultaneously. Conversely, a `dmv` can be set up so that only one protocol family can use that device. If the latter approach is taken, an SIOCSTATE ioctl must be issued by the protocol family requesting device ownership. The family's address must appear in the ''if_family'' structure member, and ''if_nomuxhdr'' must be set. Before requesting ownership, be sure to disable access to the device for all other protocol families.

The first byte of the `flags` word can be set up to indicate what mode the device should use. If bit 0 is clear, the device operates in point-to-point DDCMP mode; otherwise, it operates in maintenance mode. If bit 1 is clear, the device operates in full duplex mode; otherwise, it operates in half duplex. If bit 2 is clear, the device operates in `dmc` compatibility mode; otherwise, it operates using version 4.0 of the DDCMP protocol. The number of outstanding transmit buffers can be increased from a default of 7 to a maximum of 24 buffers by specifying a hexadecimal value in the second byte of the `flags` word. For example, if `flags` is set to 0x1800, 24 transmit buffers will be allocated on a device set up to run full duplex in `dmc` compatibility mode.

Restrictions

Note that maintenance mode should only be used to diagnose data link problems. It is not intended to be used for normal data link traffic.

If specifying maintenance mode, do not set bit 2 of the `flags` word.

Diagnostics

dmvprobe: can't start device
The `dmv` could not be started at boot time.

dmvprobe: device failed diagnostics, octal failure code = %o
The `dmv` failed diagnostics at boot time.

dmvinit: can't place dmv%d into internal loopback
Unable to place the `dmv` into internal loopback requested by user.

dmv%d: done unalloc rbuf
The device returned a receive or transmit buffer that was not allocated to it.

dmv%d: unknown address type %d
An input packet was received that contained a type of address uknown to the driver.

dmv%d bad packet address 0x%x
The device returned a buffer with an unexpected buffer address.

dmv%d: unsolicited information response: ctl = %x, data = %x
The device interrupted the driver with an information response when none was requested.

dmvd%d: bad control %o
A bad parameter was passed to the dmvload routine.

dmv%d: modem disconnect
The modem disconnected, or there was a loss of carrier while a packet was being received.

dmv%d: buffer too small
The remote node sent a packet that was too large to fit in the allocated receive buffer.

dmv%d: receive threshold reported
The dmv reported a receive threshold error.

dmv%d: transmit threshold reached
The dmv reported a transmit threshold error.

dmv%d: select threshold reached
The dmv reported a select threshold error.

dmv%d: babbling tributary reported
The dmv reported a babbling tributary error.

dmv%d: streaming tributary reported
The dmv reported a streaming tributary error.

dmv%d: MOP mode entered while DDCMP was running
dmv%d: MOP mode entered while device was halted
The dmv has entered MOP mode.

dmv%d: non existent memory reported
The dmv accessed non-existent memory.

dmv%d: device queue overflow reported
The dmv reported a queue overflow.

dmv%d: invalid counter pointer
The dmv is reporting the contents of a counter when no request was made to do so.

dmv%d: can't handle af%d
The dmv was handed a transmit message that has addresses formatted in an unsuitable address family.

dmv%d: internal loopback enable requested
The device is being put in internal loopback at a user's request.

dmv%d: internal loopback disable requested
The device is being taken out of internal loopback at a user's request.

dmvwatch: dmv%d hung, bsel0=%b, bsell = %b, bsel2=%b
The device has not responded after a long period of time.

dmz(4)

Name

dmz – DMZ32 communications interface

Syntax

device dmz0 at uba? csr 0160500 flags 0x????
vector dmzrinta dmzxinta dmzrintb dmzxintb dmzrintc dmzxintc

Description

The `dmz` device provides 24 lines of asynchronous serial line support with full modem control on all lines.

Each line attached to a DMZ32 serial line port behaves as described in `tty`(4). Input and output for each line can be set independently to run at any of 16 speeds. See `tty`(4) for the encoding.

You can specify bit *i* of `flags` for a `dmz` to say that a line is not properly connected and that the line should be treated as hardwired, with the carrier always present. For example, specifying ''flags 0x000004'' in the specification of *dmz0* would cause line 2 to be treated in this way.

Diagnostics

dmz%d: NXM line %d
No response within a timeout period from UNIBUS on a DMA transfer. This is often followed by a UNIBUS adapter error. This occurs most frequently when the UNIBUS is heavily loaded and when devices, such as RK07s, which monopolize the bus, are present. It is not serious.

dmz%d: silo overflow
The character input silo overflowed before it could be serviced. This can happen if a hard error occurs when the CPU is running with elevated priority, as the system then prints a message on the console with interrupts disabled.

Files

`/dev/tty??`	
`/dev/ttyd?`	(modem lines only)

See Also

tty(4), MAKEDEV(8)

Name

dpv – dpv data communications interface

Syntax

device dpv0 at uba0 csr 0170000

Description

The `dpv` data communications interface is used only with the 2780/3780 Terminal Emulator. It has no programmable user interface.

When a system is generated for 2780/3780 emulation on Q-bus host machines, the `dpv` data communications must be specified in the configuration file as `dpv0`.

To boot a Q-bus device having 2780/3780 emulation, the `dpv` must be specified in `rc.local`. This interface is specified by removing the comment sign from the command line already placed in the `rc.local` file:

```
#/etc/bscconfig dpv0 bsc 1
```

The `dpv` data communications interface is used with the DF126 modem.

Files

```
/etc/rc.local
/etc/bscconfig
```

See Also

2780e(1), 3780e(1), 2780d(8)

drum(4)

Name

drum – paging device

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.

Restrictions

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

Files

`/dev/drum`

See Also

MAKEDEV(8)

Name

dup – BISYNC data communications interface

Syntax

device dup0 at uba0 csr 0170000 flags 0x0f vector duprint dupxint

Description

The `dup` data communications device is used only with the 2780/3780 Terminal Emulator. It has no programmable user interface.

When a system is generated for 2780/3780 emulation on UNIBUS host machines, the `dup` data communications device must be specified in the configuration file as `dup0`.

To boot a UNIBUS device having 2780/3780 emulation, the `dup` must be specified in the `rc.local` file. This device is specified by removing the comment sign from the command line already placed in the `rc.local` file:

```
#/etc/bscconfig dup0 bsc 1
```

The `dup` data communications device is used with the DF126 modem.

Files

```
/etc/rc.local
/etc/bscconfig
```

See Also

2780e(1), 3780e(1), 2780d(8)

Name

dz – DZ11/DZ32 communications interface

Syntax

device dz0 at uba0 csr 0160100 flags 0x????
    vector dzrint dzxint

Description

A DZ11/DZ32 interface provides eight communication lines with partial modem control, adequate for dialup use. Each line attached to the DZ11/DZ32 communications interface behaves as described in `tty`(4) and can be set to run at any of 16 speeds. See `tty`(4) for the encoding.

Bit *i* of `flags` cwcanmay be specified for a `dz` to say that a line is not properly connected, and that the line should be treated as hardwired, with carrier always present. Thus, specifying ''flags 0x04'' in the specification of `dz0` would cause line 2 to be treated in this way.

The `dz` driver normally uses its input silos and polls for input at each clock tick (10 milliseconds), rather than taking an interrupt on each input character.

Diagnostics

dz%d: silo overflow
The 64-character input silo overflowed before it could be serviced. This can happen if a hard error occurs when the CPU is running with elevated priority, as the system then prints a message on the console, with interrupts disabled.

Files

`/dev/tty??`

`/dev/ttyd?`   (modem lines only)

See Also

tty(4), MAKEDEV(8)

Name

dzq – DZQ11 communications interface

Syntax

device dz0 at uba0 csr 0160100 flags 0x????
vector dzrint dzxint

Description

A DZQ11 provides four communication lines with partial modem control, adequate for dialup use. Each line attached to the DZQ11 communications interface behaves as described in `tty`(4) and can be set to run at any of 16 speeds. See `tty`(4) for the encoding.

Bit *i* of `flags` can be specified for a `dzq` to say that a line is not properly connected, and that the line should be treated as hardwired, with carrier always present. Thus, specifying ''flags 0x04'' in the specification of `dzq0` would cause line 2 to be treated in this way.

The `dzq` driver normally uses its input silos and polls for input at each clock tick (10 milliseconds), rather than taking an interrupt on each input character.

Diagnostics

dz%d: silo overflow
The 64-character input silo overflowed before it could be serviced. This can happen if a hard error occurs when the CPU is running with elevated priority, as the system then prints a message on the console with interrupts disabled.

Files

`/dev/tty??`

`/dev/ttyd?` (modem lines only)

See Also

tty(4), MAKEDEV(8)

Name

dzv – DZV11 communications interface

Syntax

device dz0 at uba0 csr 0160100 flags 0x????
vector dzrint dzxint

Description

A DZV11 provides four communication lines with partial modem control, adequate for dialup use. Each line attached to the DZV11 communications interface behaves as described in `tty`(4) and can be set to run at any of 16 speeds. See `tty`(4) for the encoding.

Bit *i* of `flags` can be specified for a `dzv` to say that a line is not properly connected, and that the line should be treated as hardwired, with carrier always present. Thus, specifying ''flags 0x04'' in the specification of `dzv0` would cause line 2 to be treated in this way.

The `dzv` driver normally uses its input silos and polls for input at each clock tick (10 milliseconds), rather than taking an interrupt on each input character.

Diagnostics

dz%d: silo overflow
The 64-character input silo overflowed before it could be serviced. This can happen if a hard error occurs when the CPU is running with elevated priority, as the system then prints a message on the console with interrupts disabled.

Files

`/dev/tty??`

`/dev/ttyd?` (modem lines only)

See Also

tty(4), MAKEDEV(8)

Name

errlog – error logging interface

Description

This is a special character device that provides an interface to the error logging daemon process, `/etc/elcsd`. This device is also accessed by other system utilities, such as the error log administration utility, `/etc/eli`.

Restrictions

This device should be readable and writable by root only, to protect access by nonsystem processes. The major number assigned to this device must correlate with the corresponding major number designation in the system kernel.

Files

`/dev/errlog`

See Also

MAKEDEV(8)

Name

fc – VAXstation serial line interface

Syntax

device fc0 at ibus? flags 0x0f vector fcxrint

Description

This serial line interface is similar to the DZQ11 four-line communications multiplexer. An `fc` interface provides four communication ports with partial modem control on port 2, adequate for dialup use. Only port 2 supports modem control (dialup access), all other ports must be operated as local lines. Each line attached to the serial line controller behaves as described in `tty`(4) and may be set to run at any of 16 speeds. For the encoding, see `tty`(4). However, configuration requirements dictate fixed speed operation of ports connected to the console terminal and graphics devices.

The `fc` ports are configured as follows:

Port	**Usage**
0	Graphics device keyboard at 9600 BPS
1	Mouse or tablet at 4800 BPS
2	Communications (with modem control)/local terminal
3	Serial printer port at 9600 BPS

A diagnostic console terminal may be connected to port 3. When the diagnostic console is in use, the processor may be halted by pressing the BREAK key. The selection of which port to use for the console is made during the processor's power on sequence and cannot be changed after power on. If the Low End Graphics Subsystem (LEGSS) console is present, it will be used; otherwise the device connected to port 3 will be the console.

For the `fc` device, the flags should always be specified as ''flags 0x0f'' (all 4 lines hardwired). The state of port 2 may be established by specifying either modem or nomodem as part of the `/etc/ttys` file entry for tty02; see `ttys`(5). The default state of port 2 may be controlled by flags bit 2. Set ''flags 0x0f'' for a hardwired line, ''flags 0x0b'' for dialup operation (wait for carrier).

The `fc` driver operates in interrupt-per-character mode (all pending characters are flushed from the silo on each interrupt). Silo alarm mode is used by the DZQ11 driver at times of high input character traffic. This mode is not used by the `fc` driver, due to the need to track mouse or tablet position changes in real time.

Restrictions

Speed must be set to 9600 BPS on the console port, 9600 BPS on the keyboard port, and 4800 BPS on the mouse port. The fc driver enforces this restriction; that is, changing speeds with the stty command may not always work on these ports.

Diagnostics

fc0: input silo overflow
The 64-character input silo overflowed before it could be serviced. This can happen if a hard error occurs when the CPU is running with elevated priority, as the system

will then print a message on the console with interrupts disabled.

Files

/dev/console	console terminal or graphics device
/dev/tty00	not used
/dev/tty01	local terminal - multiuser configuration only
/dev/tty02	dialup or local terminal
/dev/tty03	printer port or local terminal
/dev/mouse	mouse or tablet - workstation configuration only
/dev/fcscreen	console message window for workstations

See Also

console(4), devio(4), tty(4), ttys(5), MAKEDEV(8)

Name

fg – VCB03 - Midrange VAX color video subsystem

Syntax

device fg0 at ibus? flags 0x0f vector fgvint

Description

The VCB03 is a high-performance, full-page, double-buffered video subsystem capable of Z-buffering. The device consists of a 2048 bits wide x 2048 bits long x 8 or 24 plane frame buffer, a set of proprietary video chips for bitmap modification and video output, onboard VAX CPU and floating point accelerator, a 3D Transformation Engine, 1280 wide x 1024 long 19-inch color video monitor, keyboard, and a mouse or tablet.

The subsystem device driver supports a hybrid terminal with three minor devices. The first device emulates a glass tty with a screen that appears as a 80-column by 60-row page that scrolls from the bottom. This device is capable of being configured as the system console.

The second minor device number is reserved for the pointer. This device is a source of pointer state changes. (A state change is defined as an X/Y axis pointer movement or button change.) When opened, the driver couples movements of the pointer with the cursor. Pointer position changes are filtered and translated into cursor position changes in an exponential manner. Rapid movements result in large cursor position changes. All cursor positions are range checked to ensure that the cursor remains on the display.

The third minor device is opened in the raw mode by default. Opening the third device makes the driver function like a pseudo-tty in that the output destined for the first minor device is channeled to the third instead.

The Hold Screen key is supported. The driver treats this key as if CTRL/S or CTRL/Q is typed. Pressing the Hold Screen key suspends the output if it is not already suspended. The output will be resumed by pressing this key again (if the output was suspended).

Files

/dev/console
/dev/fg0
/dev/fgscreen

See Also

fc(4), ttys(5), MAKEDEV(8)

Name

hp – MASSBUS disk interface

Syntax

disk hp0 at mba0 drive 0

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 forth. The standard device names begin with 'hp' 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 with 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 or write call results in exactly one I/O operation and, 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). Similarly, `seek` calls should specify a multiple of 512 bytes.

Standard DIGITAL drive types are recognized according to the MASSBUS drive type register. The origin and size (in sectors) of the partitions on each drive are as follows:

RM03 partitions

disk	start	length	cyls
hp?a	0	15884	0-99
hp?b	16000	33440	100-308
hp?c	0	131680	0-822
hp?d	49600	15884	309-408
hp?e	65440	55936	409-758
hp?f	121440	10144	759-822
hp?g	49600	82144	309-822

RM05 partitions

disk	start	length	cyls
hp?a	0	32768	0-53
hp?b	32832	66880	54-163
hp?c	0	500384	0-822
hp?d	341696	15884	562-588
hp?e	358112	55936	589-680
hp?f	414048	86240	681-822
hp?g	341696	158592	562-822
hp?h	99712	241984	164-561

RP06 partitions

disk	start	length	cyls
hp?a	0	15884	0-37
hp?b	15884	33440	38-117
hp?c	0	340670	0-814
hp?d	49324	15884	118-155

hp?e	65208	55936	156-289
hp?f	121220	219384	290-814
hp?g	49324	291280	118-814

RM80 partitions

disk	start	length	cyls
hp?a	0	15884	0-36
hp?b	16058	33440	37-114
hp?c	0	242606	0-558
hp?d	49910	15884	115-151
hp?e	68096	55936	152-280
hp?f	125888	120559	281-558
hp?g	49910	192603	115-558

RP05 partitions

disk	start	length	cyls
hp?a	0	15884	0-37
hp?b	15884	33440	38-117
hp?c	0	171798	0-410
hp?d	2242	15884	118-155
hp?e	65208	55936	156-289
hp?f	121220	50512	290-410
hp?g	2242	122408	118-410

RP07 partitions

disk	start	length	cyls
hp?a	0	15884	0-9
hp?b	16000	66880	10-51
hp?c	0	1008000	0-629
hp?d	376000	15884	235-244
hp?e	392000	307200	245-436
hp?f	699200	308650	437-629
hp?g	376000	631850	235-629
hp?h	83200	291346	52-234

It is unwise for all of these files to be present in one installation, because there is overlap in addresses and protection becomes difficult. The hp?a partition is normally used for the root file system, the hp?b partition as a paging area, and the hp?c partition for pack-to-pack copying (it maps the entire disk). On disks larger than about 205 Megabytes, the hp?h partition is inserted prior to the hp?d or hp?g partition; the hp?g partition then maps the remainder of the pack. All disk partition tables are calculated using the `diskpart`(8) program.

Restrictions

In raw I/O, `read`(2) and `write`(2) truncate file offsets to 512-byte block boundaries, and `write` scribbles on the tail of incomplete blocks. Thus, in programs that are likely to access raw devices, `read`, `write`, and `lseek`(2) should always deal in 512-byte multiples.

Diagnostics

The following messages are printed at the console and noted in the error log file:

hp%d%c: hard error sn%d
An unrecoverable error occurred during transfer of the specified sector of the named disk partition. Either the error was unrecoverable, or a large number of retry attempts (including offset positioning and drive recalibration) could not recover the error. Additional register information may be gathered from the system error log file, `/usr/adm/syserr/syserr.<hostname>`.

hp%d: write locked
The write protect switch was set on the drive when a write was attempted. The write operation is not recoverable.

hp%d: not ready
The drive was spun down or off line when it was accessed. The I/O operation is not recoverable.

During autoconfiguration, one of the following messages may appear on the console indicating the appropriate drive type was recognized. The last message indicates the drive is of an unknown type.

The following message is written to the system error log file only:

hp%d%c: soft ecc sn%d
A recoverable ECC error occurred on the specified sector of the named disk partition. This happens normally a few times a week. If it happens more frequently than this, the sectors where the errors are occurring should be checked to see if certain cylinders on the pack or spots on the carriage of the drive or heads are indicated.

Files

```
/dev/hp???
/dev/rhp???
```

See Also

dkio(4), nbuf(4), MAKEDEV(8), uerf(8)

inet(4f)

Name

inet – Internet protocol family

Syntax

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

Description

The Internet protocol family is a collection of protocols layered atop the Internet Protocol (IP) transport layer and utilizing the Internet address format. The Internet family provides protocol support for the SOCK_STREAM, SOCK_DGRAM, and socket types. The SOCK_RAW interface provides access to the IP protocol.

Addressing

Internet addresses are 4-byte quantities, stored in network standard format (on this system, these are word- and byte-reversed). The include file <netinet/in.h defines this address as a discriminated union.

Sockets bound to the Internet protocol family utilize the following addressing structure:

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

Sockets may be created with the address INADDR_ANY to effect ''wildcard'' matching on incoming messages.

Protocols

The Internet protocol family comprises the IP transport protocol, Internet Control Message Protocol (ICMP), Transmission Control Protocol (TCP), and User Datagram Protocol (UDP). TCP is used to support the SOCK_STREAM abstraction, while UDP is used to support the SOCK_DGRAM abstraction. A raw interface to IP is available by creating an Internet socket of type SOCK_RAW. The ICMP message protocol is not directly accessible.

See Also

tcp(4p), udp(4p), ip(4p)

Name

ip – Internet Protocol

Syntax

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

s = socket(AF_INET, SOCK_RAW, 0);

Description

The IP is the transport layer protocol used by the Internet protocol family. It can be accessed through a ''raw socket'' when developing new protocols or special purpose applications. IP sockets are connectionless and are normally used with the `sendto` and `recvfrom` calls, though the `connect`(2) call can also be used to fix the destination for future packets, in which case the `read`(2) or `recv`(2) and `write`(2) or `send`(2) system calls can be used.

Outgoing packets automatically have an IP header prepended to them, based on the destination address and the protocol number the socket is created with. Likewise, incoming packets have their IP header stripped before being sent to the user.

Diagnostics

A socket operation fails with any of the following errors is returned:

[EISCONN]	Tries to establish a connection on a socket which already has one, or tries to send a datagram with the destination address specified when the socket is already connected.
[ENOTCONN]	Tries to send a datagram, but no destination address is specified and the socket has not been connected.
[ENOBUFS]	The system runs out of memory for an internal data structure.
[EADDRNOTAVAIL]	Makes an attempt to create a socket with a network address for which no network interface exists.

See Also

send(2), recv(2), inet(4f), intro(4n)

kmem (4)

Name

kmem – virtual main memory image

Description

The kmem special file is an image of the virtual main memory of the computer. It may be used, for example, to examine (and even to patch) the running system.

Byte addresses in kmem are interpreted as virtual memory addresses. For VAXs, the per-process data for the current process is at virtual address 7ffff000(16).

Restrictions

The kmem memory file is accessed one byte at a time.

Files

`/dev/kmem`

See Also

MAKEDEV(8)

Name

kUmem – UNIBUS/Q-bus virtual memory interface

Description

The `kUmem` character special file is an image of the UNIBUS/Q-bus virtual memory of the computer. It can be used, for example, to examine or patch virtual addresses mapped to UNIBUS physical space.

Byte addresses in `kUmem` are interpreted as virtual memory addresses. References to nonexistent locations cause errors to be returned.

Restrictions

The `kUmem` memory file is accessed one byte at a time.

Files

`/dev/kUmem`

See Also

MAKEDEV(8)

Name

ln – Lance Ethernet interface

Syntax

device ln0 at ibus? vector lnintr

Description

The `ln` interface provides access to a 10 Mb/s Ethernet network through the Lance controller.

The host's Internet address is specified at boot time with an SIOCSIFADDR ioctl. The `ln` interface employs the address resolution protocol described in `arp`(4p) to map dynamically between Internet and Ethernet addresses on the local network.

The interface normally tries to use a trailer encapsulation to minimize copying data on input and output. This can be disabled for an interface by setting the IFF_NOTRAILERS flag with an SIOCSIFFLAGS ioctl. Trailers are only used for packets destined for Internet hosts.

The SIOCSPHYSADDR ioctl can be used to change the physical address of the Lance. The SIOCRPHYSADDR ioctl can be used to read the physical address of the Lance.

The SIOCADDMULTI and SIOCDELMULTI ioctls can be used to add or delete multicast addresses. The Lance recognizes a maximum of 12 multicast addresses.

The SIOCRDCTRS and SIOCRDZCTRS ioctls can be used to read or ''read and clear'' the Ethernet driver counters. The argument to these two ioctls is a pointer to a counter structure, `ctrreq`, found in `<net/if.h>`.

The SIOCENABLBACK and SIOCDISABLBACK ioctls can be used to enable and disable the interface loopback mode respectively.

Diagnostics

The diagnostic error messages contain relevant information provided by the Lance.

ln%d: can't handle af%d
The interface was handed a message with addresses formated in an unsuitable address family, and the packet was dropped.

ln%d: memory error (MERR)
A memory parity error has occurred.

ln%d: lnalloc: cannot alloc memory ...
The `ln` driver was unable to allocate memory for internal data structures.

ln%d: initialization error
The `ln` driver was unable to initialize the network interface.

ln%d: SIOCADDMULTI fail, multicast list full
Too many multicast requests have been made.

See Also

arp(4p), inet(4f), intro(4n)

Name

lo – loop network interface

Syntax

pseudo-device loop

Description

The `loop` interface is a software loopback mechanism that can be used for performance analysis, software testing, and/or local communication. By default, the loopback interface is accessible at address 127.0.0.1 (nonstandard); this address may be changed with the SIOCSIFADDR ioctl.

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.

See Also

intro(4n), inet(4f)

Name

lp – LP11 line printer interface

Syntax

device lp0 at uba0 csr 0177514 flags 0x???? vector lpintr

Description

The `lp` provides the interface to any of the standard DIGITAL line printers on an LP11 parallel interface. When it is opened or closed, a suitable number of page ejects is generated. Bytes written are printed.

The unit number of the printer is specified by the minor device after removing the low 3 bits, which act as per-device parameters. Only the lowest of the low three bits is interpreted: if it is set, the device is treated as having a 64-character set, rather than a full 96-character set. In the resulting half-ASCII mode, lowercase letters are turned into uppercase and certain characters are approximated according to the following table:

Character	**Printer Approximation**
{	$\rlap{-}($
}	$\rlap{-})$
`	$\overset{\prime}{-}$
|	$\rlap{-}!$
~	$\underline{\wedge}$

The driver correctly interprets carriage returns, backspaces, tabs, and form feeds. Lines longer than the maximum page width are truncated. The default page width is 132 columns. This can be overridden by specifying, for example, ''flags 0xff'' .

Files

`/dev/lp?`

See Also

MAKEDEV(8)

Name

lpv – LPV11 parallel line printer

Syntax

device lp0 at uba0 csr 0177514 flags 0x???? vector lpintr

Description

The `lp` provides the interface to any of the standard DEC line printers on an LPV11 parallel interface. When it is opened or closed, a suitable number of page ejects is generated. Bytes written are printed.

The unit number of the printer is specified by the minor device after removing the low 3 bits, which act as per-device parameters. Only the lowest of the low three bits is interpreted: if it is set, the device is treated as having a 64-character set, rather than a full 96-character set. In the resulting half-ASCII mode, lower case letters are turned into upper case and certain characters are approximated according to the following table:

Character	**Printer Approximation**
{	$\rlap{-}($
}	$\rlap{-})$
`	$\rlap{\_}\acute{}$
|	$\rlap{-}!$
~	$\underline{\triangle}$

The driver correctly interprets carriage returns, backspaces, tabs, and form feeds. Lines longer than the maximum page width are truncated. The default page width is 132 columns. This can be overridden by specifying, for example, ''flags 0xff'' .

Files

`/dev/lp?`

See Also

MAKEDEV(8)

Name

lqf – general purpose letter quality filter

Syntax

/usr/lib/lpdfilters/lqf [**–c**] [**–n***login*] [**–h***host*] [**–w***width*] [**–l***num*] [**–i***indent*]
[accounting file]

Description

The `lqf` filter is used to filter text data destined for a letter quality printer, specifically the lqp02, but will work with any impact printer: LP25, LP26, LP27, LA50, LA75, LA100, LA120, LA210, and LG01. The filter allows escape characters and control sequences to pass through to the printer. The filter handles the device dependencies of the printers and performs accounting functions. Accounting records are written to the file specified by the **af** field in `/etc/printcap` at the time of completion for each job.

The filter can handle plain ASCII files and files that have been preprocessed by `nroff`. However, it ignores escape sequences requesting superscripting and subscripting.

The `lqf` filter can be the specified filter in both the **of** and the **if** fields in the `/etc/printcap` file. For further information, see printcap 5 . When both fields are specified, the **of** filter is used only to print the banner page. It is then stopped to allow the **if** filter access to the printer. The **if** filter maintains accounting information.

If the **of** field is the only one specified, the filter is used to print the banner page. It is then stopped and restarted. This allows the **of** filter to be used to maintain accounting information.

If the **if** field is the only one specified, the banner page will be sent directly to the printer. This is not a problem for most impact printers. For a more detailed discussion on filters see the ''Line Printer Spooler Manual'' in The Supplementary Documents.

Options

The arguments passed to the filter depend on its use. The **of** filter is called with the following arguments:

lqf –w*width* **–l***length*
The *width* and *length* values come from the **pw** and **pl** fields in the `/etc/printcap` database. The **if** (or restarted **of**) filter is passed the following arguments:

lqf –c –n*login* **–h***host* **–w***width* **–l***num* **–i***indent* **accounting file**

The **–c** flag is optional, and supplied only when control characters are to be printed, that is, when the **–l** option of lpr(1) is used to print the file. The **–w** and **–l** arguments are the same as for the **of** filter. However, they may have different values if the **–w** or **–z** options of `pr(1)` were used to print the file. The **–n** and **–h** arguments specify the login name and host name of the job owner. These arguments are used to record accounting information. The **–i** option specifies the amount of indentation to be used. The last argument is the name of the accounting file specified from the **af** field in the `/etc/printcap` database.

Diagnostics

The **lf** field (default "/dev/null") in the `/etc/printcap` database specifies error logging file name.

Files

`/etc/printcap`	Printer Capabilities Database
`/dev/lp?`	

See Also

lpr(1), pr(1), printcap(5), lpd(8), MAKEDEV(8), pac(8)
Line Printer Spooler Manual

Name

lta – lta pseudoterminal interface

Syntax

options LAT
pseudo-device lat
pseudo-device lta[*n*]

Description

The `lta` pseudoterminal interface provides support for local area transport (LAT) service. LAT service allows users to access remote nodes through the Ethernet.

To configure the LAT service for your machine, you must:

- Edit the system configuration file.
- Edit the `/etc/rc.local` file.
- Create LAT special files.
- Edit the `/etc/ttys` file.

Instructions for performing these tasks are further documented in the *Guide to Ethernet Communications Servers.*

Edit the Configuration File

Edit the configuration file to include the `LAT` option and the `lat` and `lta` pseudo-devices. The configuration file to edit is located in `/sys/conf/vax/`*HOSTNAME* or `/sys/conf/mips/`*HOSTNAME* (depending on your processor), where *HOSTNAME* is the name of your host processor, in uppercase.

The optional value for the `lta` pseudo-device entry defines the number of LAT lines to configure, a number between 1 and 256. If you do not specify a value, the default is 16 lines. For example, if you want to configure 32 LAT devices into your system, the entry for the LAT lines is:

```
pseudo-device lta32
```

To use the system as a load host for remote note maintenance functions such as loading and controlling terminal servers, you must also include an options entry for `DLI` and a pseudo-device entry for `dli` in the configuration file.

Edit the /etc/rc.local File

Edit the `/etc/rc.local` file to restart LAT service automatically when the system reboots. Add the following entry after the commands for local daemon startup:

```
if [ -f /etc/lcp ]; then
     /etc/lcp -s > /dev/console & echo -n ' lat' >/dev/console
fi
```

Create LAT Special Files

Create the LAT special files by running the MAKEDEV program from the /dev directory and specifying the lta option. You create one LAT special file for each LAT device. For example, the following MAKEDEV commands create 32 device special files for LAT devices:

```
# cd /dev
# MAKEDEV lta0
# MAKEDEV lta1
```

The option range is 1 to 7. The maximum number of LAT special files is 256.

Edit the /etc/ttys File

Edit the /etc/ttys file to include entries for all the LAT special files you created using the MAKEDEV command. For more information on how to add these entries, see ttys(5).

Files

/dev/tty??	Contains terminal devices defined to the machine.
/dev/ttyd?	Contains terminal devices defined to the machine (modem lines only).

See Also

ttys(5), MAKEDEV(8)
Guide to Ethernet Communications Servers

Name

mem – physical main memory image

Description

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

Byte addresses in `mem` are interpreted as physical memory addresses. References to nonexistent locations cause errors to be returned.

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

Restrictions

The `mem` memory file is accessed one byte at a time. This may be inappropriate for some device registers.

Files

`/dev/mem`

See Also

MAKEDEV(8)

Name

mtio – magnetic tape interface

Description

The /dev directory special files ''rmt0{l,m,h}, ..., rmt31{l,m,h''} refer to the mass storage tape drives, that may exist on several different buses depending on the bus/formatter/controller. On the BI, the TMSCP controllers, tms(4), are available. On the DSSI bus the TMSCP controller tms(4) is available. On the Q-bus the TMSCP controllers, tms(4), and the TSV05 controller, ts(4), are available. On the MASSBUS, there are the TM03, tu(4), and TM78 formatters, mu(4). On the UNIBUS, TS11 formatters, the TSU05 controller, ts(4), and the TMSCP controllers, tms(4), are available. On VAXstation 2000s and MicroVAX 2000s, the TZK50 cartridge tape subsystem, stc(4), is available. On the SCSI bus, the SCSI tapes (see tz(4)) are available. The following description applies to any mass storage tape drive.

For both the ''rewind'' and norewind special files, described later, the unit number represents a symbolic count that has no connection with the actual ''plug'' or controller number of a particular tape drive. As each tape unit special file is created, the number counts up from *0* to *31* for a total of *32* tape drives.

The special files ''rmt0l, ..., rmt31l'' are low density, ''rmt0m, ..., rmt31m'' are medium density (when a drive is ''triple density''), and ''rmt0h, ..., rmt31h'' are high density. All these special files cause a loaded and on-line tape to automatically rewind to the beginning-of-tape (BOT) when closed. Low, medium, and high density are relative to the densities supported on a particular tape drive, for example, the TS11/TSU05/TSV05 supports only 1600 bpi so its rewind namespace is ''rmt0h, ..., rmt31h''.

The special files ''nrmt0{l,m,h}, ..., nrmt31{l,m,h}'' do not cause a rewind when closed, regardless of density. When closed, the tape is positioned between two tapemarks. The norewind namespace for the TS11/TSU05/TSV05 example given above is ''nrmt0h, ..., nrmt31h''.

The rmt and nrmt special files are available to all ULTRIX utilities that can perform I/O to tape. A number of magnetic tape ioctl operations are available. The operations come under two ioctl request groups. The MTIOCTOP ioctl is used to issue tape operation commands. The MTIOCGET ioctl is used for getting status.

The **mtop** data structure defined in <sys/mtio.h> is passed as a parameter to the MTIOCTOP ioctl. The **mtop** structure is:

```
struct mtop {
        short    mt_op;
        daddr_t mt_count;
}
```

The mt_op field is used to specify the specific tape command to be performed. The mt_count field is used to specify the number of times the command is to be performed (where applicable).

The following are tape operations supported in the MTIOCTOP ioctl. These commands are specified in the mt_op field of the mtop structure.

MTWEOF Writes an end-of-file to the tape. Physically, an end of file consists of a tape mark.

MTFSF Repositions forward the number of files specified in the mt_count field. This command repositions the tape forward the specified number of tape marks. (Tape marks delimit files.) Upon successful completion of this command, the tape is physically positioned at the end of the specified number of tape marks.

MTBSF Repositions backward the number of files specified in the mt_count field. This command repositions the tape backward the specified number of tape marks. (Tape marks delimit files.) Upon successful completion of the command, the tape is physically positioned at the beginning of the specified number of tape marks. Note that, due to the difference in the side of a tape mark that a reposition command leaves the tape positioned, the MTFSF and MTBSF commands are not strictly reciprocal operations. For example, if a tape is initially positioned at the bottom of tape (BOT) and the command MTFSF 1 is issued followed by the command MTBSF 1, the tape does not return to the BOT position. Instead, the tape is positioned on the BOT side of the first tape mark.

MTFSR Repositions forward the number of records specified in the mt_count field. This command returns a failure if a tape mark is encountered. This error condition indicates that there were not as many records remaining in the file as specified by the mt_count parameter.

MTBSR Repositions backward the number of records specified in the mt_count field. This command returns a failure if a tape mark is encountered. This error condition indicates that there were not as many records between the present position and the beginning of the file as specified in the mt_count parameter.

MTREW Rewinds the tape. This command repositions to the beginning of the tape.

MTOFFL Rewinds and unloads the tape.

MTNOP Does not perform any tape operation. Always returns success from a tape file.

MTCACHE
Enables the use of controller-based write-back caching. Some tape controllers support caching as a performance enhancement. Caching can speed up tape transfer operations by keeping the unit streaming more effectively. When using cached mode of operation, the MTFLUSH command should be used to flush cached data to media. See the description of MTFLUSH for more details.

MTNOCACHE
Disables use of the controller's write-back cache. This mode of

operation can result in performance degradation over cached mode.

MTCSE	Clears serious exception. Certain operations cause the tape unit to go into a serious exception state. An example of this is when the physical end-of-media foil is encountered. Typically, when a tape is in serious exception state, all data transfer operations fail. In order to acknowledge this exception condition and to allow further operations to proceed, this command is provided.
MTFLUSH	Flushes the controller's write-back cache. This command is intended to be used in conjunction with the MTCACHE command. When caching has been enabled using the MTCACHE command, writes to the tape will receive completion status when the data has been transferred to the controller's write-back cache. In the unlikely event of controller error, it is possible that the data will not be transferred to the physical media. To insure data integrity, the MTFLUSH command is provided to force a flush of the cache to physical media. Failure of this command with *errno* set to ENXIO means that the drive does not support the flush command. Failure with *errno* set to EIO indicates that the cache flush has failed. In this case, the application should retry writing records that have been written since the last successful MTFLUSH command.

The global variable *errno* is set to ENXIO if the command specified in mt_op is not recognized or not supported by the respective tape driver.

Each `read` or `write` system call reads or writes the next record on the tape. In the case of `write`, the record has the same length as the buffer given. During a `read`, the record size is passed back as the number of bytes read, provided it is no greater than the buffer size. If the record is long, an error is returned. Seeks are ignored. Positioning is done with a tape ioctl call. When n-buffered I/O is not being used (see `nbuf`(4)), a zero byte count is returned when a tape mark is read, but another read fetches the first record of the next tape file. When n-buffered I/O is being used (see `nbuf`(4)), a zero byte count is returned when a tape mark is read, but another read will not fetch the first record of the next tape file. In this situation, all outstanding read requests return a status of 0. In order to allow reading to proceed to the next file, the MTCSE command must be issued. When a file open for writing is closed, two end-of-files (EOF) are written. If a tape reaches the end-of-tape (EOT) marker, the ENOSPC *errno* value is set.

Each `read` or `write` system call causes the file offset associated with the device special file to be incremented. This file offset is reset to 0 when the file is closed. If a program does an unusually large number or reads and writes to the tape, it is possible to cause the file offset to be incremented beyond the maximum allowable value. When this happens, any further `read` or `write` system calls fail with an error number of EINVAL. This situation can only occur if the tape is read or written to several times over, using repositioning commands such as MTREW to reposition backwards on the tape. It is recommended that any application which expects to make numerous passes over the tape use the `lseek` system call to reset the file offset to zero, for example, lseek(fd,0,0)

Restrictions

For SCSI tapes on VAX systems, the maximum tape record length is limited to 16K bytes (K = 1024).

For SCSI tapes on both VAX and RISC systems, the MTCACHE, MTNOCACHE, and MTFLUSH ioctls are not supported.

Files

```
/dev/rmt???
/dev/nrmt???
```

See Also

lseek(2), mu(4), scsi(4), stc(4), tms(4), ts(4), tu(4), tz(4), MAKEDEV(8)

mu(4)

Name

mu – TM78/TU78/9 magnetic tape interface

Syntax

master mt0 at mba? drive ?
tape mu0 at mt0 slave 0

Description

Prior to Version 2.0, this device was referenced by `mt`(4).

The TM78/TU78/9 combination provides a standard tape drive interface as described in `mtio`(4). Only 1600 and 6250 bpi are supported; the TU78/9 runs at 125 ips and autoloads tapes.

Restrictions

Only the raw tape is supported.

Diagnostics

mu%d: no write ring.
An attempt was made to write on the tape drive when no write ring was present.

mu%d: not on line.
An attempt was made to access the tape while it was off line.

mu%d: hard error bn%d.
A tape error occurred at block *bn*. When possible, the driver will have retried the operation that failed several times before reporting the error. Additional register information may be gathered from the system error log file,
`/usr/adm/syserr/syserr.<hostname>`.

mu%d: blank tape.
An attempt was made to read a blank tape (a tape without even end-of-file marks).

mu%d: off line.
During an I/O operation, the device was set off line.

Files

/dev/rmt???
/dev/nrmt???

See Also

mtio(4), nbuf(4), MAKEDEV(8), uerf(8)

Name

nbuf – select multiple-buffer operation to a raw device

Syntax

#include <sys/ioctl.h>

ioctl(d, FIONBUF, count)
int d;
int *count;

status=ioctl(d, FIONBDONE, buffer)
int d, status;
char **buffer;

Description

The I/O operations to raw devices are usually performed through a single buffer. This means that the issuing process must wait for a buffer to complete before the process can do anything else. An *N-buffered* I/O operation allows a process to begin an I/O operation and continue doing something else until the operation has finished. Once *N-buffered* operation is enabled, `read`(2) and `write`(2) acts as before except that buffer completion is not guaranteed when the call returns. If the operation starts without errors, `read`(2) and `write`(2) return as if the operation were successful. That is, the number of requested bytes have transferred and file pointers are updated. On read operations, the process must not use the contents of the started buffer until the buffer actually completes. On write operations, the process must not reuse the buffer until the operation actually completes. A second `ioctl` is used to check the status of previously issued *N-buffered* read/write requests to determine when the operation has really completed.

N-buffered I/O is used through a set of `ioctl` calls. Setting the *request* argument in an `ioctl` call to FIONBUF enables *count* buffers to be used with the raw device associated with the file descriptor *d*. If *count* is zero, the N-buffered operation is terminated and any pending buffers are completed. A *count* less than zero is invalid. Any started I/O buffer's status is checked by the `ioctl` call with the *request* argument set to FIONBDONE, with the address of the buffer used as an argument. The status field returns the actual byte count transferred or any error encountered on the I/O operation. The FIONBDONE ioctl must be called before re-using a buffer. FIONBDONE blocks the process until the given buffer completes (unless FNDELAY has been specified with `fcntl`(2), at which point EWOULDBLOCK is returned). In addition, a signal can be generated whenever a buffer completes, if FIOASYNC has been specified with `fcntl`(2).

The `select`(2) call is also useful in checking on the status of pending buffers. The `select`(2) call returns immediately if less than *count* operations have been started on an N-buffered channel. Otherwise, `select` blocks the specified amount of time for a buffer to become done. At this point, FIONBDONE must be used to return actual status of the pending buffer.

Diagnostics

The `ioctl` call fails if one or more of the following are true:

[EBADF]	The *d* argument is not a valid descriptor.
[ENOTTY]	The *d* argument is not associated with a character special device.
[ENOTTY]	The specified request does not apply to the kind of object which the descriptor *d* references.
[EINVAL]	The *request* or *argp* argument is not valid. Returned for FIONBDONE, if requested buffer was never started. Also returned for FIONBUF, if this device does not support *N-buffered* I/O.

See Also

fcntl(2), ioctl(2), select(2)

Name

ni – BVP DEBNT/NI interface

Syntax

```
controller  aie?      at vaxbi? node?
device      bvpni0    at aie? vector bvpniintr
```

Description

The `ni` driver provides access to a 10 Mb/s Ethernet network through the NI port of the DEBNT controller. The DEBNT also has a tape port that shares the controller structure with the NI port.

The host's Internet address is specified at boot time with an SIOCSIFADDR ioctl. The `ni` driver employs the Address Resolution Protocol described in `arp`(4p) to dynamically map addresses on the local network between Internet and Ethernet.

The driver normally tries to use a ''trailer'' encapsulation to minimize copying data on input and output. This can be disabled for an interface by setting the IFF_NOTRAILERS flag with an SIOCSIFFLAGS ioctl. Trailers are only used for packets destined for Internet hosts.

The SIOCSPHYSADDR ioctl can be used to change and SIOCRPHYSADDR can be used to read the physical address of the NI. SIOCADDMULTI and SIOCDELMULTI can be used to add or delete multicast addresses. The NI supports a maximum of 32 multicast addresses. The argument to the latter ioctls is a pointer to an ifreq structure found in <net/if.h>.

SIOCCRDCTRS and SIOCRDZCTRS ioctls can be used to ''read'' or ''read and clear'' network counters. The argument to the latter two ioctls is a pointer to a counter structure ''ctrreq found in <net/if.h>.

The ioctls SIOCENABLBACK and SIOCDISABLBACK can be used to enable and disable the interface loopback mode respectively.

Restrictions

The PUP protocol family is not supported.

Diagnostics

ni%d in wrong state
DEBNT is unable to be attached during autoconfiguration time, because it is not in the undefined state.

ni%d cannot initialize
DEBNT failed to become initialized after a request to become initialized.

nid%d cannot enable
DEBNT failed to become enabled after a request to become enabled.

reset ni%d %x %x %x %x
DEBNT has requested a software reset. Values of port control, port status, port error, and port data are given to help identify what caused the reset request.

ni%d SUME error detected %x %x %x %x
Some error has been detected. Values of port error, port data, port status, and port control are given to help identify the cause of the error.

ni%d cant handle af%d
A packet with an undefined protocol type has been sent to DEBNT.

ni%d multi failed, multicast list full
Too many multicast requests have been made.

See Also

arp(4p), inet(4f), intro(4n)

Name

null – data sink

Description

Data written on a `null` special file is discarded.

Reads from a `null` special file always return 0 bytes.

Files

`/dev/null`

See Also

MAKEDEV(8)

Name

packetfilter, ip – Ethernet packet filter

Syntax

options PACKETFILTER
pseudo-device packetfilter

Description

The packet filter pseudo-device driver provides a raw interface to Ethernets and similar network data link layers. Packets received that are not used by the kernel (for example, to support the IP and DECnet protocol families) are available through this mechanism. The packet filter driver is kernel-resident code provided by the ULTRIX operating system. The driver appears to applications as a set of character special files, one for each open packet filter application. (Throughout this reference page, the word *file* refers to such a character special file.)

You create the minor device files with the `MAKEDEV(8)` script using these commands:

```
# cd /dev
# MAKEDEV pfilt
```

A single call to `MAKEDEV` with an argument of `pfilt` creates 64 character special files in `/dev/pf`, which are named `pfilt`*nnn*, where *nnn* is the unit number. Successive calls to `MAKEDEV` with arguments of `pfilt1`, `pfilt2`, and `pfilt3` make additional sets of 64 sequentially numbered packet filters to a maximum of 256. The maximum number of packet filter special files files is limited to 256, which is the maximum number of minor device numbers allowed for each major device number. (See `MAKEDEV(8)` for more information on making system special files.)

For opening these special files, the ULTRIX operating system provides the `pfopen(3)` library routine.

Associated with each open instance of a packet filter special file is a user-settable packet filter ''program'' that is used to select which incoming packets are delivered by that packet filter special file. Whenever a packet is received from the net, the packet filter driver successively applies the filter programs of each of the open packet filter files to the packet, until one filter program ''accepts'' the packet. When a filter accepts the packet, it is placed on the packet input queue of the associated special file. If no filters accept the packet, it is discarded. The format of a packet filter is described later.

Reads from these files return the next packet from a queue of packets that have matched the filter. If the read operation specifies insufficient buffer space to store the entire packet, the packet is truncated and the trailing contents lost. Writes to these files transmit packets on the network, with each write operation generating exactly one packet.

The packet filter supports a variety of different Ethernet data-link levels:

10Mb Ethernet
Packets consist of fourteen or more bytes, with the first six bytes specifying the destination Ethernet address, the next six bytes the source Ethernet address, and the next two bytes specifying the packet type. (This is the Standard Ethernet.)

3Mb Ethernet
Packets consist of four or more bytes, with the first byte specifying the source Ethernet address, the second byte specifying the destination Ethernet address, and the next two bytes specifying the packet type. (On the network, the source and destination addresses are in the opposite order.)

Byte-swapping 3Mb Ethernet
Packets consist of four or more bytes, with the first byte specifying the source Ethernet address, the second byte specifying the destination Ethernet address, and the next two bytes specifying the packet type. Each short word (pair of bytes) is swapped from the network byte order. This device type is provided only for backwards-compatibility.

The remaining words are interpreted according to the packet type. Note that 16-bit and 32-bit quantities may have to be byteswapped (and possibly short-swapped) to be intelligible on an ULTRIX system.

The packet filters treat the entire packet, including headers, as uninterpreted data. The user must supply the headers for transmitted packets (although the system makes sure that the source address is correct) and the headers of received packets are delivered to the user. The packet filter mechanism does not know anything about the data portion of the packets it sends and receives.

In addition to the FIONREAD `ioctl` request (described in the `tty`(4) reference page), the application can apply several special `ioctl` requests to an open packet filter file. The calls are divided into five categories: packet-filter specifying, packet handling, device configuration, administrative, and miscellaneous.

Packet-filter Specification ioctl Request

The `EIOCSETF` `ioctl` is central to the operation of the packet filter interface, because it specifies which packets the application wishes to receive. It is used to set the packet filter ''program'' for an open packet filter file, and is of the form:

```
ioctl(fildes, EIOCSETF, filter)
struct enfilter *filter
```

The `enfilter` stsructure is defined in `<net/pfilt.h>` as:

```
struct enfilter
{
       u_char   enf_Priority;
       u_char   enf_FilterLen;
       u_short  enf_Filter[ENMAXFILTERS];
};
```

A packet filter consists of a priority, the filter command list length (in shortwords), and the filter command list itself. Each filter command list specifies a sequence of actions that operate on an internal stack. Each shortword of the command list specifies an action and a binary operator.

Command List Actions

The action can be one of the following:

ENF_PUSHLIT
Pushes the next shortword of the command list on the stack.

ENF_PUSHWORD+N
Pushes shortword N of the incoming packet on the stack.

ENF_PUSHZERO
Pushes a zero. Is slightly faster than ENF_PUSHLIT with an explicit literal.

ENF_PUSHONE
Pushes a one. Is slightly faster than ENF_PUSHLIT with an explicit literal.

ENF_PUSHFFFF
Pushes 0xFFFF. Is slightly faster than ENF_PUSHLIT with an explicit literal.

ENF_PUSH00FF
Pushes 0x00FF. Is slightly faster than ENF_PUSHLIT with an explicit literal.

ENF_PUSHFF00
Pushes 0xFF00. Is slightly faster than ENF_PUSHLIT with an explicit literal.

ENF_NOPUSH
Defined as zero.

Binary Operators

When both an action and an operator are specified in the same shortword, the action is performed, followed by the operation. You can combine an action with an operator using bitwise OR; for example,

```
((ENF_PUSHWORD+3) | ENF_EQ)
```

The binary operator, which can be one of the following, operates on the top two elements of the stack and replaces them with its result:

ENF_EQ	Returns true if the result is equal.
ENF_NEQ	Returns true if the result is not equal.
ENF_LT	Returns true if the result is less than.
ENF_LE	Returns true if the result is less than or equal.
ENF_GT	Returns true if the result is greater than.
ENF_GE	Returns true if the result is greater than or equal.
ENF_AND	Returns the result of the binary AND operation.
ENF_OR	Returns the result of the binary OR operation.

ENF_XOR	Returns the result of the binary XOR operation.
ENF_NOP	Defined as zero.
ENF_CAND	Returns false immediately if the result is false, and continues execution of the filter otherwise. (Short-circuit operator)
ENF_COR	Returns true immediately if the result is true, and continues execution of the filter otherwise. (Short-circuit operator)
ENF_CNAND	Returns true immediately if the result is false, and continues execution of the filter otherwise. (Short-circuit operator)
ENF_CNOR	Returns false immediately if the result is true, and continues execution of the filter otherwise. (Short-circuit operator)

The short-circuit operators are so called because they terminate the execution of the filter immediately if the condition they are checking for is found, and continue otherwise. All the short-circuit operators pop two elements from the stack and compare them for equality. Unlike the other binary operators, these four operators do not leave a result on the stack, even if they continue.

Use the short-circuit operators whenever possible, to reduce the amount of time spent evaluating filters. When you use them, you should also arrange the order of the tests so that the filter will succeed or fail as soon as possible. For example, checking a word in an address field of an Ethernet packet is more likely to indicate failure than the Ethernet type field.

The special action ENF_NOPUSH and the special operator ENF_NOP can be used only to perform the binary operation or only to push a value on the stack. Because both are defined to be zero, specifying only an action actually specifies the action followed by ENF_NOP, and specifying only an operation actually specifies ENF_NOPUSH followed by the operation.

After executing the filter command list, a nonzero value (true) left on top of the stack (or an empty stack) causes the incoming packet to be accepted for the corresponding packet filter file and a zero value (false) causes the packet to be passed through the next packet filter. 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.

To resolve problems with overlapping or conflicting packet filters, the filters for each open packet filter file are ordered by the driver according to their priority (lowest priority is 0, highest is 255). When processing incoming packets, filters are applied according to their priority (from highest to lowest) and for identical priority values according to their relative ''busyness'' (the filter that has previously matched the most packets is checked first), until one or more filters accept the packet or all filters reject it and it is discarded.

Normally once a packet is delivered to a filter, it is not presented to any other filters. However, if the packet is accepted by a filter in nonexclusive mode (ENNONEXCL set using EIOCMBIS, described in the following section), the packet is passed along to lower-priority filters and may be delivered more than once. The use of nonexclusive filters imposes an additional cost on the system, because it increases the average number of filters applied to each packet.

The packet filter for a packet filter file is initialized with length 0 at priority 0 by open(2), and hence, by default, accepts all packets in which no higher-priority filter is interested.

Priorities should be assigned so that, in general, the more packets a filter is expected to match, the higher its priority. This prevents a lot of checking of packets against filters that are unlikely to match them.

The filter in this example accepts incoming RARP (Reverse Address Resolution Protocol) broadcast packets.

The filter first checks the Ethernet type of the packet. If it is not a RARP (Reverse ARP) packet, it is discarded. Then, the RARP type field is checked for a reverse request (type 3), followed by a check for a broadcast destination address. Note that the packet type field is checked before the destination address, because the total number of broadcast packets on the network is larger than the number of RARP packets. Thus, the filter is ordered with a minimum amount of processing overhead.

```
struct enfilter f =
{
    36, 0,                  /* priority and length */
    ENF_PUSHWORD + 6,
    ENF_PUSHLIT, 0x3580,
    ENF_CAND,               /* Ethernet type == 0x8035 (RARP) */
    ENF_PUSHWORD + 10,
    ENF_PUSHLIT, 0x0300,
    ENF_CAND,               /* reverse request type = 0003 */
    ENF_PUSHWORD + 0,
    ENF_PUSHLIT, 0xFFFF,
    ENF_CAND,               /* dest addr = FF-FF */
    ENF_PUSHWORD + 1,
    ENF_PUSHLIT, 0xFFFF,
    ENF_CAND,               /* dest addr = FF-FF */
    ENF_PUSHWORD + 2,
    ENF_PUSHLIT, 0xFFFF,
    ENF_EQ                  /* dest addr = FF-FF */
};
```

Note that shortwords, such as the packet type field, are in network byte-order. The literals you compare them to may have to be byte-swapped on machines like the VAX.

By taking advantage of the ability to specify both an action and operation in each word of the command list, you could abbreviate the filter to the following:

```
struct enfilter f =
{
    36, 0,                          /* priority and length */
    ENF_PUSHWORD + 6,
    ENF_PUSHLIT | ENF_CAND,
    0x3580,                         /* Ethernet type == 0x8035 (RARP) */
    ENF_PUSHWORD + 10,
    ENF_PUSHLIT | ENF_CAND,
    0x0300,                         /* reverse request type = 0003 */
    ENF_PUSHWORD + 0,
    ENF_PUSHFFFF | ENF_CAND,        /* dest addr = FF-FF */
    ENF_PUSHWORD + 1,
    ENF_PUSHFFFF | ENF_CAND,        /* dest addr = FF-FF */
    ENF_PUSHWORD + 2,
    ENF_PUSHFFFF | ENF_EQ           /* dest addr = FF-FF */
};
```

Packet-Handling ioctl Requests

These `ioctl` requests control how the packet filter processes input packets and returns them to the application process. The most useful of these requests set and clear so-called ''mode bits'' for the file and are of this form:

```
ioctl(fildes, code, bits)
u_short *bits;
```

In these calls, *bits* is a bitmask specifying which bits to set or clear. The applicable *codes* are:

EIOCMBIS
Sets the specified mode bits.

EIOCMBIC
Clears the specified mode bits.

The *bits* are:

ENTSTAMP
If set, a received packet is preceded by a header structure (see the description of `enstamp` following) that includes a time stamp and other information.

ENBATCH
If clear, each `read`(2) system call returns at most one packet. If set, a `read` call might return more than one packet, each of which is preceded by an `enstamp` header.

ENPROMISC
If set, this filter will be applied to promiscuously-received packets. This puts the interface into ''promiscuous mode'' only if this has been allowed by the superuser using the EIOCMAXBACKLOG `ioctl` call (described later).

ENNONEXCL
If set, packets accepted by this filter will be available to any lower-priority filters. If clear, no lower-priority filter will see packets accepted by this filter.

ENHOLDSIG
If clear, means that the driver should disable the effect of EIOCENBS (described later) once it has delivered a signal. If set (the default), the effect of EIOCENBS persists.

The `enstamp` structure contains useful information about the packet that immediately follows it; in ENBATCH mode, it also allows the reader to separate the packets in a batch. It is defined in `<net/pfilt.h>` as:

```
struct enstamp {
       u_short          ens_stamplen;
       u_short          ens_flags;
       u_short          ens_count;
       u_short          ens_dropped;
       u_long           ens_ifoverflows;
       struct           timevalens_tstamp;
};
```

The fields are:

ens_stamplen
The length of `enstamp` structure in bytes. The packet data follows immediately.

ens_flags
Indicates how the packet was received. The *bits* are:

ENSF_PROMISC
Received promiscuously (unicast to some other host).

ENSF_BROADCAST
Received as a broadcast.

ENSF_MULTICAST
Received as a multicast.

ENSF_TRAILER
Received in a trailer encapsulation. The packet has been rearranged into header format.

ens_count
The length of the packet in bytes (does not include the `enstamp` header).

ens_dropped
The number of packets accepted by this filter but dropped because the input queue was full; this is a cumulative count since the previous `enstamp` was read from this packet filter file. This count may be completely wrong if the ENNONEXCL mode bit is set for this filter.

ens_ifoverflows
The total number of input overflows reported by the network interface since the system was booted.

ens_tstamp
The approximate time the packet was received.

If the buffer returned by a batched `read`(2) contains more than one packet, the offset from the beginning of the buffer at which each `enstamp` structure begins is an integer multiple of the word-size of the processor. For example, on a VAX, each `enstamp` is aligned on a longword boundary (provided that the buffer address passed to the `read`(2) system call is aligned). The alignment (in units of bytes) is given by the constant `ENALIGNMENT`, defined in `<net/pfilt.h>`. If you have an integer *x*, you can use the macro `ENALIGN(`*x*`)` to get the least integer that is a multiple of `ENALIGNMENT` and not less than *x*. For example, this code fragment reads and processes one batch:

```
char *buffer = &(BigBuffer[0]);
int buflen;
int pktlen, stamplen;
struct enstamp *stamp;

buflen = read(f, buffer, sizeof(BigBuffer));
while (buflen > 0) {
    stamp = (struct enstamp *)buffer;
    pktlen = stamp->ens_count;
    stamplen = stamp->ens_stamplen;
    ProcessPacket(&(buffer[stamplen]), pktlen);    /* your code here */
    if (buflen == (pktlen + stamplen))
        break;                                   /* last packet in batch */
    pktlen = ENALIGN(pktlen);      /* account for alignment padding */
    buflen -= (pktlen + stamplen);
    buffer += (pktlen + stamplen);            /* move to next stamp */
}
```

If a buffer filled by a batched read contains more than one packet, the final packet is never truncated. If, however, the entire buffer is not big enough to contain a single packet, the packet will be truncated; this is also true for unbatched reads. Therefore, the buffer passed to the `read`(2) system call should always be big enough to hold the largest possible packet plus an `enstamp` structure. (See the `EIOCDEVP ioctl` request later in this reference page for information on how to determine the maximum packet size. See also the `EIOCTRUNCATE ioctl` request for an example that delivers only the desired number of bytes of a packet.)

Normally, a packet filter application blocks in the `read` system call until a received packet is available for reading. There are several ways to avoid blocking indefinitely: an application can use the `select(2)` system call, it can set a ''timeout'' for the packet filter file, or it can request the delivery of a signal (see `sigvec`(2)) when a packet matches the filter.

EIOCSETW
The packet filter interface limits the number of packets that can be queued for delivery for a specific packet filter file. Application programs can vary this ''backlog'', if necessary, using the following call:

`ioctl(`*fildes*`, EIOCSETW, `*maxwaitingp*`)`
`u_int *`*maxwaitingp*`;`

The pointer *maxwaitingp* points to an integer containing the input queue size to be set. If this is greater than the maximum allowable size (see EIOCMAXBACKLOG later), it is set to the maximum. If it is zero, it is set to a default value.

EIOCFLUSH

After changing the packet filter program, the input queue may contain packets that were accepted under the old filter. To flush the queue of incoming packets, use the following:

ioctl (*fildes*, EIOCFLUSH, 0)

EIOCTRUNCATE

An application, such as a network load monitor, that does not want to see the entire packet can ask the packet filter to truncate received packets at a specified length. This action may improve performance by reducing data movement.

To specify truncation, use:

ioctl (*fildes*, EIOCTRUNCATE, *truncationp*)
u_int **truncationp*;

The pointer *truncationp* points to an integer specifying the truncation length, in bytes. Packets shorter than this length are passed intact.

This example, a revision of the previous example, illustrates the use of EIOCTRUNCATE, which causes the packet filter to deliver only the first *n* bytes of a packet, not the entire packet.

```
char *buffer = &(BigBuffer[0]);
int buflen;
int pktlen, stamplen;
struct enstamp *stamp;
int truncation = SIZE_OF_INTERESTING_PART_OF_PACKET;

if (ioctl(f, EIOCTRUNCATE, &truncation) < 0)
    exit(1);

while (1) {
    buflen = read(f, buffer, sizeof(BigBuffer));
    while (buflen > 0) {
       stamp = (struct enstamp *)buffer;
       pktlen = stamp->ens_count;     /* ens_count is untruncated length */
       stamplen = stamp->ens_stamplen;

       ProcessPacket(&(buffer[stamplen]), pktlen);     /* your code here */

       if (pktlen > truncation)       /* truncated portion not in buffer */
           pktlen = truncation;
       if (buflen == (pktlen + stamplen))
           break;                                /* last packet in batch */
       pktlen = ENALIGN(pktlen);        /* account for alignment padding */
       buflen -= (pktlen + stamplen);
       buffer += (pktlen + stamplen);              /* move to next stamp */
   }
}
```

Two calls control the timeout mechanism; they are of the following form:

#include <net/time.h>

ioctl (*fildes*, *code*, *tvp*)

struct timeval **tvp*;

The *tvp* argument is the address of a `struct timeval` containing the timeout interval (this is a relative value, not an absolute time). The codes are:

EIOCGRTIMEOUT
Returns the current timeout value.

EIOCSRTIMEOUT
Sets the timeout value. When the value is positive, a `read`(2) call returns a 0 if no packet arrives during the period. When the timeout value is zero, reads block indefinitely (this is the default). When the value is negative, a `read`(2) call returns a 0 immediately if there are no queued packets. Note that the largest legal timeout value is a few million seconds.

Two calls control the signal-on-reception mechanism; they are of the following form:

```
ioctl(fildes, code, signp)
u_int *signp;
```

The *signp* argument is a pointer to an integer containing the number of the signal to be sent when an input packet arrives. The applicable *codes* are:

EIOCENBS
Enables the specified signal when an input packet is received for this file. If the ENHOLDSIG flag (see EIOCMBIS later) is not set, further signals are automatically disabled whenever a signal is sent to prevent nesting, and hence must be explicitly re-enabled after processing. When the signal number is 0, this call is equivalent to EIOCINHS.

EIOCINHS
Disables signaling on packet reception. The pointer *signp* is ignored. This is the default when the file is first opened.

Device Configuration ioctl Requests

EIOCIFNAME
Each packet filter file is associated with a specific network interface. To find out the name of the interface underlying the packet filter file, use the following:

```
#include <net/socket.h>
#include <net/if.h>

ioctl(fildes, EIOCIFNAME, ifr)
struct ifreq *ifr;
```

The interface name (for example, ''de0'') is returned in *ifr->ifr_name*; other fields of the *struct ifreq* are not set.

EIOCSETIF
To set the interface associated with a packet filter file, use the following:

```
ioctl(fildes, EIOCSETIF, ifr)
struct ifreq *ifr;
```

The interface name should be passed *ifr->ifr_name*; other fields of the *struct ifreq* are ignored. The name provided may be one of the actual interface names, such as ''de0'*U or qe1'', or it may be a pseudo-interface name of the form ''pf*n*'', used to specify the *n*th interface attached to the system. For example, ''pf0'' specifies the first interface. This is useful for applications that do not know the names of specific interfaces. Pseudo-interface names are never returned by EIOCIFNAME.

EIOCDEVP

To get device parameters of the network interface underlying the packet filter file, use the following:

```
ioctl(fildes, EIOCDEVP, param)
struct endevp *param;
```

The `endevp` structure is defined in `<net/pfilt.h>` as:

```
struct endevp {
       u_char   end_dev_type;
       u_char   end_addr_len;
       u_short  end_hdr_len;
       u_short  end_MTU;
       u_char   end_addr[EN_MAX_ADDR_LEN];
       u_char   end_broadaddr[EN_MAX_ADDR_LEN];
};
```

The fields are:

end_dev_type	Specifies the device type: ENDT_3MB, ENDT_BS3MB, or ENDT_10MB.
end_addr_len	Specifies the address length in bytes (for example, 1 or 6).
end_hdr_len	Specifies the total header length in bytes (for example, 4 or 14).
end_MTU	Specifies the maximum packet size, including header, in bytes.
end_addr	The address of this interface; aligned so that the low order byte of the address is in *end_addr[0]*.
end_broadaddr	The hardware destination address for broadcasts on this network.

Administrative ioctl Requests

EIOCMAXBACKLOG

The maximum queue length that can be set using EIOCSETW depends on whether the process is running as the superuser or not. If so, the maximum is a kernel constant; otherwise, the maximum is a value that can be set, by the superuser, for each interface. To set the maximum non-superuser backlog for an interface, use EIOCSETIF to bind to the interface, and then use the following:

```
ioctl(fildes, EIOCMAXBACKLOG, maxbacklogp)
int *maxbacklogp;
```

The pointer *maxbacklogp* points to an integer containing the maximum value. (If *maxbacklogp* points to an integer containing a negative value, it is replaced with the current backlog value, and no action is taken.)

EIOCALLOWPROMISC

Certain kinds of network-monitoring applications need to place the interface in ''promiscuous mode'', where it receives all packets on the network. Promiscuous mode can be set by the superuser with the */etc/ifconfig* command, or the superuser can configure an interface to go into promiscuous mode automatically if any packet filter applications have the ENPROMISC mode bit set. To do so, use EIOCSETIF to bind to the interface, and then use the following:

```
ioctl(fildes, EIOCALLOWPROMISC, allowp)
int *allowp;
```

The pointer *allowp* points to an integer containing a Boolean value (nonzero means promiscuous mode is set automatically). (If *allowp* points to an integer containing a negative value, it is replaced with the current Boolean value, and no action is taken.)

EIOCMFREE

To find out how many packet filter files remain for opening, use this `ioctl`, which places the number in the integer pointed to by *mfree* :

```
ioctl(fildes, EIOCMFREE, mfree)
int *mfree;
```

Miscellaneous ioctl Requests

Two calls are provided for backwards compatibility and should not be used in new code. These calls are used to set and fetch parameters of a packet filter file (*not* the underlying device; see EIOCDEVP). The forms for these call are:

```
#include <sys/types.h>
#include <net/pfilt.h>

ioctl(fildes, code, param)

struct eniocb *param;
```

The structure `eniocb` is defined in `<net/pfilt.h>` as:

```
struct eniocb
{
        u_char  en_addr;
        u_char  en_maxfilters;
        u_char  en_maxwaiting;
        u_char  en_maxpriority;
        long    en_rtout;
};
```

The applicable *codes* are:

EIOCGETP

Fetch the parameters for this file.

EIOCSETP

Set the parameters for this file.

All the fields, which are described later, except *en_rtout*, are read-only.

en_addr	No longer maintained; use EIOCDEVP.
en_maxfilters	The maximum length of a filter command list; see EIOCSETF.

en_maxwaiting	The maximum number of packets that can be queued for reading on the packet filter file; use EIOCMAXBACKLOG.
en_maxpriority	The highest allowable filter priority; see EIOCSETF.
en_rtout	The number of clock ticks to wait before timing out on a read request and returning a zero length. If zero, reads block indefinitely until a packet arrives. If negative, read requests return a zero length immediately if there are no packets in the input queue. Initialized to zero by `open`(2), indicating no timeout. (Use EIOCSRTIMEOUT and EIOCGRTIMEOUT.)

Restrictions

Because the packet filter include file `<net/pfilt.h>` was originally named `<sys/enet.h>`, some filter applications may need to be updated.

A previous restriction against accessing data words past approximately the first hundred bytes in a packet has been removed. However, it becomes slightly more costly to examine words that are not near the beginning of the packet.

Because packets are streams of bytes, yet the filters operate on short words, and standard network byte order is usually opposite from VAX byte order, the relational operators ENF_LT, ENF_LE, ENF_GT, and ENF_GE are not particularly useful. If this becomes a severe problem, a byte-swapping operator could be added.

Files

`/dev/pf/pfilt`*nnn*	Packet filter special files

See Also

pfopen(3), de(4), ln(4), ni(4), qe(4), xna(4), ifconfig(8), MAKEDEV(8), pfconfig(8c), pfstat(8)
The Packet Filter: An Efficient Mechanism for User-Level Network Code

Name

pm – monochrome/color bitmap graphics

Syntax

device pm0 at ibus? vector pmvint

Description

The video subsystem provides a half page or full page, user-accessible bitmap display for graphics. The subsystem consists of a 256 Kbytes (monochrome) or a 1 Mbyte (color) block of dual port RAM, a mouse or tablet, a keyboard, and a video monitor.

The subsystem device driver supports a hybrid terminal with three minor devices. The first minor device emulates a glass tty with a screen that appears as an 80-column by 56-row page that scrolls from the bottom. This device is capable of being configured as the system console.

The second minor device is reserved for the mouse. This device is a source of mouse state changes. (A state change is defined as an X/Y axis mouse movement or button change.) When opened, the driver couples movements of the mouse with the cursor. Mouse position changes are filtered and translated into cursor position changes in an exponential manner. Rapid movements result in large cursor position changes. All cursor positions are range checked to ensure that the cursor remains on the display.

The third minor device provides an access path for console output that does not disturb the graphics display. The caller can open the device `/dev/xcons`. When this device is open, the graphics driver redirects console device output to the input buffer of this device. This mechanism disables console output on the screen and saves the output for later display. This preserves the graphic display integrity.

Input and output on the first and third minor devices are processed by the standard line disciplines.

The Hold Screen key is supported. The graphics driver treats this key as if CTRL/S or CTRL/Q had been pressed. Pressing the Hold Screen key suspends the output (if it is not already suspended). To resume the output, press the Hold Screen key again.

Files

`/dev/console`	Console terminal or graphics device
`/dev/mouse`	Mouse or tablet graphics device
`/dev/xcons`	Console message window for workstation

See Also

console(4), devio(4), tty(4), ttys(5), MAKEDEV(8)

Name

pty – pseudoterminal driver

Syntax

pseudo-device pty[*n*]

Description

The `pty` driver provides support for a device-pair termed a *pseudoterminal.* A pseudoterminal is a pair of character devices, a *master* device and a *slave* device. The slave device provides processes with an interface identical to that described in `tty`(4). However, whereas all other devices that provide the interface described in `tty`(4) have a hardware device of some sort behind them, the slave device has, instead, another process manipulating it through the master half of the pseudoterminal. That is, anything written on the master device is given to the slave device as input and anything written on the slave device is presented as input on the master device. The slave device can be opened multiple times, while the master half can be opened only once.

If no optional *n* value is given defining the number of pseudoterminal pairs to be configured, 16 pseudoterminal pairs are configured. All pseudoterminal lines should have a corresponding entry in the /etc/ttys file. This must be done to insure that logins that use pseudoterminals will be tracked in the utmp and wtmp files.

The following `ioctl` calls apply only to pseudoterminals:

TIOCSTOP
: Stops output to a terminal (for example, like typing CTRL/S). Takes no parameter.

TIOCSTART
: Restarts output (stopped by TIOCSTOP or by typing CTRL/S). Takes no parameter.

TIOCPKT
: Enable/disable *packet* mode. Packet mode is enabled by specifying (by reference) a nonzero parameter and disabled by specifying (by reference) a zero parameter. When applied to the master side of a pseudoterminal, each subsequent `read` from the terminal will return data written on the slave part of the pseudoterminal 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 by typing CTRL/S.

  TIOCPKT_START
  : whenever output to the terminal is restarted.

TIOCPKT_DOSTOP
: whenever the stop character is CTRL/S and the start character is CTRL/Q.

TIOCPKT_NOSTOP
: whenever the start and stop characters are not CTRL/S and/or CTRL/Q.

This mode is used by `rlogin`(1c) and `rlogind`(8c) to implement a remote-echoed, locally flow-controlled (using CTRL/S or CTRL/Q, or both) remote login with proper back-flushing of output. It can be used by other similar programs.

TIOCREMOTE
: A mode for the master half of a pseudoterminal, independent of TIOCPKT. This mode causes input to the pseudoterminal to be flow controlled and not input edited (regardless of the terminal mode). Each write to the control terminal produces a record boundary for the process reading the terminal. 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 end-of-file character. TIOCREMOTE can be used when doing remote line editing in a window manager, or whenever flow-controlled input is required.

TIOCMASTER
: Allows the master to have complete control over the pseudoterminal and causes the slave side to sleep until the master relinquishes control. This is useful in preventing changes on the pseudoterminal from going undetected and being reset by the master.

Files

`/dev/pty??`	(master pseudoterminals)
`/dev/tty??`	(slave pseudoterminals)

See Also

tty(4), MAKEDEV(8)

qd(4)

Name

qd – VCB02 (QDSS) video subsystem

Syntax

device qd0 at uba0 csr 0177400 flags 0x0f vector qddint qdaint qdiint

Description

A VCB02 provides a half-page or full-page user-accessible bit map display for graphics applications. The device consists of a 256kb Q22 bus memory array, a 15-inch or 19-inch video monitor, and a VX10X-EA mouse.

The subsystem device driver supports a hybrid terminal with three minor devices. The first device emulates a glass tty with a screen that appears as a 120-column by 80-row page that scrolls from the bottom. This device is capable of being configured as the system console.

The second minor device is opened in the raw mode by default. Opening the second device makes the driver function like a pseudoterminal in that the output destined for the first minor device is channeled to the second instead. Input and output on the first two minor device numbers are processed by the standard line disciplines.

The third minor device number is reserved for the mouse. This device is a source of mouse state changes. (A state change is defined as an X/Y axis mouse movement or button change.) When opened, the driver couples movements of the mouse with the cursor. Mouse position changes are filtered and translated into cursor position changes in an exponential manner. Rapid movements result in large cursor position changes. All cursor positions are range checked to ensure that the cursor remains on the display.

If there is a VCB02 module at the standard address, the system will use it as the system console. All input/output destined for `/dev/console` will use the VCB02 instead. (This is done by overwriting the device switch tables.) There is a second set of device switch entries configured for the console that can be used as an additional terminal or printer port by making a special device file using major number 38 and minor number 0 and making the appropriate entry in `/etc/ttys`.

Files

```
/dev/qd?
/dev/qconsole
```

See Also

tty(4), ttys(5), MAKEDEV(8)

Name

qe – DEQNA/DELQA Ethernet interface

Syntax

device qe0 at uba0 csr 0174440 vector qeintr

Description

The `qe` interface provides access to a 10 Mb/s Ethernet network through a DEQNA/DELQA controller.

The host's Internet address is specified at boot time with an SIOCSIFADDR ioctl. The `qe` interface employs the address resolution protocol described in `arp`(4p) to dynamically map between Internet and Ethernet addresses on the local network.

The interface normally tries to use a ''trailer'' encapsulation to minimize copying data on input and output. This can be disabled for an interface by setting the IFF_NOTRAILERS flag with an SIOCSIFFLAGS ioctl. Trailers are only used for packets destined for Internet hosts.

The SIOCSPHYSADDR ioctl can be used to change and SIOCRPHYSADDR can be used to read the physical address of the board. SIOCADDMULTI and SIOCDELMULTI can be used to add or delete multicast addresses. The board recognizes at most 10 multicast addresses. The argument to the latter ioctls is a pointer to an ifreq structure found in <net/if.h>.

SIOCRDCTRS and SIOCRDZCTRS ioctls can be used to read or ''read and clear'' the board counters. The argument to the latter two ioctls is a pointer to a counter structure "ctrreq" found in `<net/if.h>`.

The ioctls SIOCENABLBACK and SIOCDISABLBACK can be used to enable and disable the interface loopback mode.

Restrictions

The PUP protocol family is not supported.

Diagnostics

Various error messages can occur while transmitting or receiving packets. For example,

qe%d: can't handle af%d
The interface was handed a message with addresses formatted in an unsuitable address family; the packet was dropped.

See Also

arp(4p), inet(4f), intro(4n)

Name

qv – VCB01 (QVSS) video subsystem

Syntax

device qv0 at uba0 csr 0177200 flags 0x0f vector qvkint qvvint

Description

A VCB01 provides a half-page or full-page, user-accessible bitmap display for graphics applications. The device consists of a 256 kbyte Q22 bus memory array, a 15-inch or 19-inch video monitor, and a VX10X-EA mouse.

The subsystem device driver supports a hybrid terminal with three minor devices. The first device emulates a glass tty with a screen that appears as a 120-column by 80-row page that scrolls from the bottom. This device is capable of being configured as the system console.

The second minor device is opened in the raw mode by default. Opening the second device makes the driver function like a pseudoterminal in that the output destined for the first minor device is channeled to the second instead. Input and output on the first two minor device numbers are processed by the standard line disciplines.

The third minor device number is reserved for the mouse. This device is a source of mouse state changes. (A state change is defined as an X/Y axis mouse movement or button change.) When opened, the driver couples movements of the mouse with the cursor. Mouse position changes are filtered and translated into cursor position changes in an exponential manner. Rapid movements result in large cursor position changes. All cursor positions are range checked to ensure that the cursor remains on the display.

If there is a VCB01 module at the standard address, the system will use it as the system console. All input/output destined for `/dev/console` will use the VCB01 instead. This is done by overwriting the device switch tables. There is a second set of device switch entries configured for the console that can be used as an additional terminal or printer port by making a special device file using major number 38 and minor number 0 and making the appropriate entry in `/etc/ttys`.

Restrictions

The use of the bitmap as source or destination of I/O operations is not supported. Minor devices 1 and 2 are read only and are the only ones supported by the MAKEDEV script. Shared access by multiple processes is not constrained or supported. The system only supports one VCB01. The mouse device buffers the last 50 events (state changes). If the console port has been enabled as an additional terminal it must be disabled before removing the VCB01 to avoid two login processes on the same device.

Files

```
/dev/qvscreen
/dev/mouse
```

See Also

tty(4), ttys(5), MAKEDEV(8)

Name

ra – MSCP disk interface

Syntax

For UNIBUS, Q-bus:
 controller uda0 at uba?
 controller uq0 at uda0 csr 0172150 vector uqintr
 disk ra0 at uq0 drive 0

For VAX BI:
 controller kdb0 at vaxbi0 node 4
 controller uq0 at kdb0 vector uqintr
 disk ra0 at uq0 drive 0
 controller bvpssp0 at aiol vector bvpsspintr
 disk ra0 at bvpssp0 drive 0

For MSI Bus:
 adapter msi0 at nexus?
 controller dssc0 at msi0 msinode 0
 disk ra0 at dssc0 drive 3

For VAX CI/HSC:
 adapter ci0 at nexus?
 adapter ci0 at vaxbi? node?
 controller hsc0 at ci0 cinode 6
 disk ra0 at hsc0 drive 3

Description

Prior to Version 2.0, this device was referenced by `uda(4)`.

This is a driver for all DIGITAL MSCP disk controllers. All controllers communicate with the host through a packet-oriented protocol termed the Mass Storage Control Protocol (MSCP).

The following rules are used to determine the major and minor numbers that are associated with an `ra` type disk. There is a range of major numbers used to represent `ra` disks. Each major number represents 32 disks. For this reason, the first major number associated with `ra` disks represents logical unit number 0 through logical unit number 31. Similarly the second major number represents logical unit number 32 through logical unit number 63. The minor number is used to represent both the logical unit number and partition. A disk partition refers to a designated portion of the physical disk. To accomplish this, the 8-bit minor number is broken up into two parts. The low three bits of the minor number specify a disk partition. These three bits allow for the naming of eight partitions. The partitions are named a,b,c,d,e,f,g and h. The upper five bits of the minor number specify the logical unit number within a group of 32 disks.

The device special file names associated with `ra` disks are based on the following conventions, which are closely associated with the minor number assigned to the disk. The standard device names begin with `ra` for the block special file and `rra` for the raw (character) special file. Following the `ra` is the logical unit number and then a letter, a through h, to represent the partition. Throughout this reference page, the question mark (?) character represents the logical unit number in the name of the device special file. For example ra?b could represent ra0b, ra1b, and so on.

The following examples illustrate how the logical unit number is calculated given the major and minor number of an `ra` disk. For the device special file rra6a, the major number is 60 and the minor number is 48. The partition is represented by the low 3 bits of the number 48. The low 3 bits will be 0 which specifies the "a" partition. The upper 5 bits of 48 specifies the number 6. The major number is 60. Because 60 is the base major number, it represents the first group of 32 disks. For this reason, there is no need to adjust the unit number for a high order grouping. Putting all these pieces together reveals that the major/minor pair 60/48 refers to the "a" partition of logical unit 6. As another example, the following computation determines the logical unit number corresponding to the major/minor pair 62,49. The low 3 bits of the minor number gives the number 1, which is the "b" partition. The upper 5 bits of the minor number gives the number 6. The major number is 62. Subtracting 62 from the base major number of 60 gives a value of 2. This means that 2 groups of 32 disks preceed the unit in question. For this reason, the logical unit number is as follows: (2 * 32) + 6 = 70. The figure 6 is from the minor number. Therefore, the major/minor pair 62,49 refers to the "b" partition of logical unit number 70, or rra70b.

The disk can be accessed through either the block special file or the character special file. The block special file accesses the disk using the file system's normal buffering mechanism. Reads and writes to the block special file can specify any size. This avoids the need to limit data transfers to the size of physical disk records and to calculate offsets within disk records. The file system may break up large read and write requests into smaller fixed size transfers to the disk.

The character special file provides a raw interface which allows for direct transmission between the disk and the user's read or write buffer. In contrast to the block special file, reads and writes to the raw interface must be done on full sectors only. For this reason, in raw I/O, counts should be multiples of 512 bytes (a disk sector). In the same way, `seek` calls should specify a multiple of 512 bytes. A single read or write to the raw interface results in exactly one I/O operation, consequently raw I/O may be considerably more efficient for large transfers. Multiply buffered I/O operations are possible to any raw MSCP device. (See `nbuf`(4) for more information.)

Disk Support

This driver handles all disk drives that may be connected to an MSCP-based controller. Consult the *ULTRIX Software Product Description* to determine which controllers are supported for which CPU types and hardware configurations.

The starting location and length (in 512-byte sectors) of the disk partitions of each drive are shown in the following table. Partition sizes can be changed by `chpt`(8). For further information, see `dkio`(4).

RA60 partitions

disk	start	length
ra?a	0	32768
ra?b	32768	50160
ra?c	0	400176
ra?d	24298	52416
ra?e	295344	52416

ra?f	347760	52415
ra?g	82928	160000
ra?h	24928	157247

RA70 partitions

disk	start	length
ra?a	0	32768
ra?b	32768	66690
ra?c	0	547042
ra?d	0	99458
ra?e	0	281805
ra?f	99458	447583
ra?g	99458	182347
ra?h	281805	265236

RA80 partitions

disk	start	length
ra?a	0	32768
ra?b	32768	50160
ra?c	0	237212
ra?d	82928	51428
ra?e	134356	51428
ra?f	185784	51428
ra?g	82928	154284
ra?h	0	0

RA81 partitions

disk	start	length
ra?a	0	32768
ra?b	32768	66690
ra?c	0	891072
ra?d	323840	210538
ra?e	46996	210538
ra?f	680534	210538
ra?g	99458	160000
ra?h	259458	631614

RA82 partitions

disk	start	length
ra?a	0	32768
ra?b	32768	66690
ra?c	0	1216665
ra?d	99458	220096
ra?e	319554	219735
ra?f	539289	437760
ra?g	99458	877591
ra?h	977049	239616

RA90 partitions

disk	start	length
ra?a	0	32768
ra?b	32768	127072
ra?c	0	2409680
ra?d	159840	420197
ra?e	580037	420197
ra?f	1000234	840393
ra?g	159840	1680787
ra?h	1840627	535526

RA92 partitions

disk	start	length
ra?a	0	32768
ra?b	32768	127072
ra?c	0	2940951
ra?d	159840	420197
ra?e	580037	420197
ra?f	1000234	840393
ra?g	159840	1680787
ra?h	1840627	1100324

RD31 partitions

disk	start	length
ra?a	0	15884
ra?b	15884	10024
ra?c	0	41560
ra?d	0	0
ra?e	0	0
ra?f	0	0
ra?g	25908	15652
ra?h	0	0

RD32 partitions

disk	start	length
ra?a	0	15884
ra?b	15884	15625
ra?c	0	83236
ra?d	31509	25863
ra?e	57372	25864
ra?f	0	0
ra?g	31509	51727
ra?h	0	0

RD51 partitions

disk	start	length
ra?a	0	15884
ra?b	15884	5716
ra?c	0	21600
ra?d	0	0
ra?e	0	0
ra?f	0	0
ra?g	0	0
ra?h	0	0

RD52 partitions

disk	start	length
ra?a	0	15884
ra?b	15884	9766
ra?c	0	60480
ra?d	0	0
ra?e	0	50714
ra?f	50714	9766
ra?g	25650	34830
ra?h	15884	44596

RD53 partitions

disk	start	length
ra?a	0	32768
ra?b	32768	50160
ra?c	0	138672
ra?d	0	0
ra?e	0	0
ra?f	0	0
ra?g	82928	55744
ra?h	32768	105904

RD54 partitions

disk	start	length
ra?a	0	32768
ra?b	32768	50160
ra?c	0	311200
ra?d	82928	130938
ra?e	213866	97334
ra?f	0	0
ra?g	82928	228272
ra?h	0	0

RF30 partitions

disk	start	length
ra?a	0	32768
ra?b	32768	50160
ra?c	0	293040
ra?d	82928	130938
ra?e	213866	79173
ra?f	0	0
ra?g	82928	210111
ra?h	0	0

RF31 partitions

disk	start	length
ra?a	0	32768
ra?b	32768	66690
ra?c	0	744400
ra?d	0	99458
ra?e	0	281805
ra?f	99458	644942
ra?g	99458	182347
ra?h	281805	462595

RF71 partitions

disk	start	length
ra?a	0	32768
ra?b	32768	66690
ra?c	0	781440
ra?d	0	99458
ra?e	0	281805
ra?f	99458	681982
ra?g	99458	182347
ra?h	281805	499635

RRD40 (read only) partitions

disk	start	length
ra?a	0	0
ra?b	0	0
ra?c	0	1171875
ra?d	0	0
ra?e	0	0
ra?f	0	0
ra?g	0	0
ra?h	0	0

RRD50 (read only) partitions

disk	start	length
ra?a	0	15884
ra?b	15884	33440
ra?c	0	1171875
ra?d	131404	122993
ra?e	254397	122993
ra?f	377390	794485
ra?g	49324	82080
ra?h	131404	1040471

RX33 partitions

disk	start	length
ra?a	0	2400
ra?b	0	0
ra?c	0	2400
ra?d	0	0
ra?e	0	0
ra?f	0	0
ra?g	0	0
ra?h	0	0

RX50 partitions

disk	start	length
ra?a	0	800
ra?b	0	0
ra?c	0	800
ra?d	0	0
ra?e	0	0
ra?f	0	0
ra?g	0	0
ra?h	0	0

ESE20 partitions

disk	start	length
ra?a	0	15884
ra?b	15884	33440
ra?c	0	245760
ra?d	49324	130938
ra?e	180262	65498
ra?f	0	0
ra?g	49324	196436
ra?h	0	0

Usually the ra?a partition is used for the root file system, the ra?b partition as a paging area. The ra?c partition for pack to pack copying because it maps the entire disk.

Files

```
/dev/ra???
/dev/rra???
```

See Also

nbuf(4), dkio(4), chpt(8), MAKEDEV(8), uerf(8)

Name

rb – IDC/RL02 disk interface

Syntax

controller idc0 at uba? csr 0175606 vector idcintr
disk rb0 at idc0 drive 0

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 forth. The standard device names begin with `rb` followed by the drive number and then a letter, a through h, for partitions 0 through 7. The question mark (?) character stands here for a drive number in the range 0 through 7.

The block files access the disk by the system's normal buffering mechanism and can 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 results in exactly 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 additional letter r, for example, `rrx2c`.

Although RL02 disks have 256-byte sectors, the driver emulates 512-byte sectors. Raw I/O counts should be multiples of 512 bytes (a normal disk sector). In the same way, `seek` calls should specify a multiple of 512 bytes.

The origin and size (in 512-byte sectors) of the pseudodisks on each drive are as follows:

RL02 partitions:

disk	start	length	cyl
rb?a	0	15884	0-397
rb?b	15884	4520	398-510
rb?c	0	20480	0-511
rb?d	15884	4520	398-510
rb?g	0	20480	0-511

Restrictions

In raw I/O, `read` and `write` functions truncate file offsets to 512-byte block boundaries; `write` overwrites the tail of incomplete blocks. Thus, in programs that are likely to access raw devices, `read`(2), `write`(2), and `lseek`(2) should always deal in 512-byte multiples.

Diagnostics

The following messages can appear at the console:

rb%%d%c: hard error sn%d
An unrecoverable error occurred during transfer of the specified sector of the specified disk partition. Either the error was unrecoverable, or a large number of retry attempts (including offset positioning and drive recalibration) could not recover

the error. Additional register information can be gathered from the system error log file, `/usr/adm/syserr/syserr.`*<hostname>*.

rb%d: write protected
The write protect switch was set on the drive when a write was attempted. The write operation is not recoverable.

idc%d: lost interrupt
A timer watching the controller detected no interrupt for an extended period while an operation was outstanding. This indicates a hardware or software failure. The error causes a UNIBUS reset and retry of the pending operations. If the controller continues to lose interrupts, this error will recur a few seconds later.

Files

```
/dev/rb???
/dev/rrb???
```

See Also

dkio(4), nbuf(4), MAKEDEV(8)

Name

rd – RD31, RD32, RD53, RD54 Small VAX Winchester disks

Description

The `rd` Winchester disks, when used either by the workstation or multiuser configurations of the busless Small VAX processor, are supported by the SDC disk driver. The SDC driver handles RD31, RD32, RD53, and RD54 disks on drives 0 and 1.

Files

```
/dev/rd[0-1][a-f]
/dev/rrd[0-1][a-f]
```

See Also

sdc(4)

Name

rk – RK711/RK07 disk interface

Syntax

controller hk0 at uba? csr 0177440 vector rkintr
disk rk0 at hk0 drive 0

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 forth. The standard device names begin with ''hk'' followed by the drive number and then a letter, a through h, for partitions 0 through 7. The question mark (?) character stands here for a drive number in the range 0 through 7.

The block files access the disk using the system's normal buffering mechanism and can 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 or write call results in exactly 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 additional letter r, for example, `rrx2c`.

In raw I/O, counts should be multiples of 512 bytes (a disk sector). In the same way, `seek` calls should specify a multiple of 512 bytes.

The origin and size (in sectors) of the pseudodisks on each drive are as follows:

RK07 partitions:

disk	start	length	cyl
rk?a	0	15884	0-240
rk?b	15906	10032	241-392
rk?c	0	53790	0-814
rk?g	26004	27786	393-813

Restrictions

In raw I/O, `read` and `write` functions truncate file offsets to 512-byte block boundaries; `write` overwrites the tail of incomplete blocks. Thus, in programs that are likely to access raw devices, `read`(2), `write`(2), and `lseek`(2) should always deal in 512-byte multiples.

Diagnostics

The following messages are printed at the console:

rk%d%c: hard error sn%d
An unrecoverable error occurred during transfer of the specified sector of the specified disk partition. Either the error was unrecoverable, or a large number of retry attempts (including offset positioning and drive recalibration) could not recover the error. Additional register information can be gathered from the system error log file, `/usr/adm/syserr/syserr.`*<hostname>*.

rk%d: write locked
The write protect switch was set on the drive when a write was attempted. The write operation is not recoverable.

rk%d: not ready
The drive was spun down or off line when it was accessed. The I/O operation is not recoverable.

rk%d: not ready (came back!)
The drive was not ready. But, after printing this message (which takes a fraction of a second), it was ready. The operation is recovered, if no further errors occur.

hk%d: lost interrupt
A timer watching the controller detected no interrupt for an extended period while an operation was outstanding. This indicates a hardware or software failure. Spinning down drives while they are being accessed causes this error to occur. The error causes a UNIBUS reset and retry of the pending operations. If the controller continues to lose interrupts, this error will recur a few seconds later.

rk%d%c: soft ecc sn%d
This message is written to the system error log file only. A recoverable ECC error occurred on the specified sector in the specified disk partition. This happens normally a few times a week. If it happens more frequently than this, the sectors where the errors are occurring should be checked to see if the same physical location on the disk pack is causing the error. Errors in the same area on the disk pack indicate the pack is going bad. Random errors can be caused by a pack going bad or a pending hardware problem.

Files

```
/dev/rk???
/dev/rrk???
```

See Also

dkio(4), nbuf(4), MAKEDEV(8), uerf(8)

Name

rl – RL211/RL02 disk interface

Syntax

controller hl0 at uba? csr 0174400 vector rlintr
disk rl0 at hl0 drive 0

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 forth. The standard device names begin with `rl` followed by the drive number and then a letter, a through h, for partitions 0-7. The question mark (?) character stands here for a drive number in the range 0-7.

The block files access the disk by the system's normal buffering mechanism and can 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 results in exactly 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 addition letter r, for example, `rrx2c`.

Although RL02 disks have 256-byte sectors, the driver emulates 512-byte sectors. Raw I/O counts should be multiples of 512 bytes (a normal disk sector). In the same way, `seek` calls should specify a multiple of 512 bytes.

The origin and size (in 512-byte sectors) of the pseudodisks on each drive are as follows:

RL02 partitions:

disk	start	length	cyl
rl?a	0	15884	0-397
rl?b	15884	4520	398-510
rl?c	0	20480	0-511
rl?d	15884	4520	398-510
rl?g	0	20480	0-511

Restrictions

In raw I/O, `read` and `write` functions truncate file offsets to 512-byte block boundaries, and `write` overwrites the tail of incomplete blocks. Thus, in programs that are likely to access raw devices, `read`(2), `write`(2), and `lseek`(2) should always deal in 512-byte multiples.

Diagnostics

The following messages are printed at the console:

rl%d%c: hard error sn%d
An unrecoverable error occurred during transfer of the specified sector of the specified disk partition. Either the error was unrecoverable, or a large number of retry attempts (including offset positioning and drive recalibration) could not recover

the error. Additional register information may be gathered from the system error log file, `/usr/adm/syserr/syserr.`*<hostname>*.

rl%d: write protected
The write protect switch was set on the drive when a write was attempted. The write operation is not recoverable.

hl%d: lost interrupt
A timer watching the controller detected no interrupt for an extended period while an operation was outstanding. This indicates a hardware or software failure. The error causes a UNIBUS reset and retry of the pending operations. If the controller continues to lose interrupts, this error will recur a few seconds later.

Files

```
/dev/rl???
/dev/rrl???
```

See Also

dkio(4), nbuf(4), MAKEDEV(8), uerf(8)

Name

rx – RX33/RX23 Small VAX floppy disk drive

Description

The RX33 floppy disk drive, when used by either the workstation or multiuser configurations of the busless Small VAX processor, is supported by the SDC disk driver. The SDC disk driver also supports the RX23 floppy disk drive in VAXstation 3100 Model 30 systems. The SDC driver handles the RX33/RX23 disk on drive 2.

The RX33 floppy disk drive reads and writes both RX33 and RX50 floppy diskettes. The RX23 floppy disk drive reads and writes high density (18 sectors per track) RX23 floppy diskettes and reads double density (9 sectors per track) RX23 floppy diskettes.

Formatting RX23 Floppy Diskettes (VAXstation 3100 Model 30 Only)

The floppy diskettes used with the RX23 floppy disk drive are not always preformatted. Although the ULTRIX system does not currently include support for online formatting of diskettes, the VAXstation 3100 diagnostic `t 70` command allows you to format diskettes.

Use the following steps to format diskettes:

1. Shut down the ULTRIX Worksystem Software, and halt the processor.

   ```
   # /etc/shutdown -h +5
   ```

2. Insert the diskette into the RX23 drive.
3. Enter the following at the prompt:

   ```
   >>> t 70
   ```

   The `t 70` command begins to prompt you for information. Answer each query as in the following example. Press the RETURN key after each response:

   ```
        KA42 RDRXfmt

   VSfmt_QUE_unitno (0-3) ? 2

   VSfmt_QUE_RXmedtyp

   ( 1=RX33 ),( 2=RX23 ) ? 2

   VSfmt_QUE_RUsure (DUA2 1/0)   ? 1

   VSfmt_STS_FMTing ..... OK

   VSfmt_STS_CKRXfmt .....OK

   VSfmt_RES_Succ
   >>>
   ```

Files

```
/dev/rx2a
/dev/rx2c
/dev/rrx2a
/dev/rrx2c
```

See Also

sdc(4)

Name

rz – SCSI disk interface

Syntax

```
VAX NCR 5380:
   adapter      uba0     at nexus?
   controller   scsi0    at uba0      csr 0x200c0080  vector szintr
   disk         rz0      at scsi0     drive 0
VAX DEC SII:
   adapter      ibus0    at nexus?
   controller   sii0     at ibus?     vector sii_intr
   disk         rz0      at sii0      drive 0
RISC DEC SII:
   adapter      ibus0    at nexus?
   controller   sii0     at ibus?     vector sii_intr
   disk         rz0      at sii0      drive 0
RISC NCR ASC:
   adapter      ibus0    at nexus?
   controller   asc0     at ibus?     vector ascintr
   disk         rz0      at asc0      drive 0
```

Description

The `rz` driver is for all Digital SCSI disk drives.

The following rules are used to determine the major and minor numbers that are associated with an `rz` type disk. There is one major number used to represent `rz` disks. The major number represents 32 disks. The minor number is used to represent the both the SCSI unit number and partition. A disk partition refers to a designated portion of the physical disk. To accomplish this, the 8-bit minor number is broken up into two parts. The low three bits of the minor number specify a disk partition. These three bits allow for the naming of eight partitions. The partitions have a letter, a through h, as their name. The upper five bits of the minor number specify the SCSI unit number within a group of 32 disks.

The device special file names associated with `rz` disks are based on the following conventions. These conventions are closely associated with the minor number assigned to the disk. The standard device names begin with `rz` for the block special file and `rrz` for the raw (character) special file. Following the `rz` is the logical unit number and then a letter, a through h, to represent the partition. Throughout this reference page, the question mark (?) character represents the logical unit number in the name of the device special file. For example, rz?b could represent rz0b, rz1b, and so on.

The following examples illustrate how the SCSI unit number is calculated given the major and minor number of an `rz` disk. For the device special file rrz6a, the major number is 56 and the minor number is 48. The partition is represented by the lower three bits of the number 48. The lower three bits are 0, which specifies the "a" partition. The upper five bits of 48 specify the number 6. The major number is 56. Because 56 is the base major number, it represents the group of 32 disks. Putting all these pieces together reveals that the major/minor pair 56/48 refers to the ''a'' partition of SCSI unit 6.

The disk can be accessed through either the block special file or the character special file. The block special file accesses the disk using the file system's normal buffering mechanism. Reads and writes to the block special file can specify any size. This avoids the need to limit data transfers to the size of physical disk records and to calculate offsets within disk records. The file system can break up large read and write requests into smaller fixed size transfers to the disk.

The character special file provides a raw interface that allows for direct transmission between the disk and the user's read or write buffer. In contrast to the block special file, reads and writes to the raw interface must be done on full sectors only. For this reason, in raw I/O, counts must be a multiple of 512 bytes (a disk sector). In the same manner, `seek` calls must specify a multiple of 512 bytes. A single read or write to the raw interface results in exactly one I/O operation. Consequently raw I/O may be considerably more efficient for large transfers. Multiply buffered I/O operations are possible to any raw SCSI device. (See `nbuf`(4) for more information.)

For systems with SCSI disks, the first boot of the ULTRIX software after the system is powered on may take longer than expected. This delay is normal and is caused by the software spinning up the SCSI disk drives.

Disk Support

This driver handles all disk drives that can be connected to the SCSI bus. Consult the *ULTRIX Software Product Description* to determine which drives are supported for which CPU types and hardware configurations.

The starting location and length (in 512 byte sectors) of the disk partitions of each drive are shown in the following table. Partition sizes can be changed by `chpt`(8). For further information, see `dkio`(4).

RZ22 partitions

disk	start	length
rz?a	0	32768
rz?b	32768	69664
rz?c	0	102431
rz?d	0	0
rz?e	0	0
rz?f	0	0
rz?g	0	0
rz?h	0	0

RZ23 partitions

disk	start	length
rz?a	0	32768
rz?b	32768	66690
rz?c	0	204864
rz?d	99458	35135
rz?e	134593	35135
rz?f	169728	35136
rz?g	99458	105406
rz?h	134593	70271

RZ24 partitions

disk	start	length
rz?a	0	32768
rz?b	32768	131072
rz?c	0	409792
rz?d	163840	81984
rz?e	245824	81984
rz?f	327808	81984
rz?g	163840	245952
rz?h	0	0

RZ55 partitions

disk	start	length
rz?a	0	32768
rz?b	32768	131072
rz?c	0	649040
rz?d	163840	152446
rz?e	316286	152446
rz?f	468732	180308
rz?g	163840	485200
rz?h	0	0

RZ56 partitions

disk	start	length
rz?a	0	32768
rz?b	32768	131072
rz?c	0	1299174
rz?d	163840	292530
rz?e	456370	292530
rz?f	748900	550273
rz?g	163840	1135334
rz?h	731506	567668

RZ57 partitions

disk	start	length
rz?a	0	32768
rz?b	32768	184320
rz?c	0	2025788
rz?d	831488	299008
rz?e	1130496	299008
rz?f	1429504	596284
rz?g	217088	614400
rz?h	831488	1194300

RRD40 (read only) partitions

disk	start	length
rz?a	0	(size varies per CD)
rz?b	0	0
rz?c	0	(size varies per CD)
rz?d	0	0
rz?e	0	0
rz?f	0	0
rz?g	0	0
rz?h	0	0

RX23 partitions

disk	start	length
rz?a	0	2879
rz?b	0	0
rz?c	0	2879
rz?d	0	0
rz?e	0	0
rz?f	0	0
rz?g	0	0
rz?h	0	0

Usually, the rz?a partition is used for the root file system and the rz?b partition as a paging area. The rz?c partition is used for disk-to-disk copying because it maps the entire disk.

Files

```
/dev/rz???
/dev/rrz???
```

See Also

nbuf(4), dkio(4), SCSI(4), chpt(8), MAKEDEV(8), uerf(8)

Name

scs – Systems Communications Services network interface.

Syntax

pseudo device scsnet

Description

The `scs` interface provides network access to any network device supported by the System Communications Services subsystem (SCS). Currently, the only network device available is the Computer Interconnect (CI).

The host's Internet address is specified at boot time with an SIOCSIFADDR ioctl.

Restrictions

Only the Internet protocol family (TCP/IP/UDP) is supported. The host numbers must be between 1 and 15 inclusive. Zero is reserved.

See Also

inet(4f), intro(4n)

Name

SCSI – Small Computer System Interconnect

Description

The ULTRIX system interfaces to disk and tape devices through the Small Computer System Interconnect (SCSI). Initial ULTRIX SCSI support is limited to the Digital-supplied mass storage devices. The following devices are fully supported on the ULTRIX system:

- Winchester disks: RZ22, RZ23, RZ24, RZ55, RZ56, RZ57, RX23
- Magnetic tapes: TZ30, TZK50, TLZ04, TSZ05
- Optical disks: RRD40

Under the ULTRIX operating system, a SCSI device is referred to by its logical name. Logical names take the following form:

```
nn#
```

The *nn* argument is the two-character name; the number sign (#) represents the unit number. The two character names for SCSI devices are:

rz	- RZ22, RZ23, RZ24, RZ55, RZ56, RZ57, and RRD40 disks
tz	- TZ30, TZK50, TLZ204, and TSZ05 tapes

The unit number is a combination of the SCSI bus number, either 0, 1, ... and the device's target ID number. The unit number is eight times the bus number plus the target ID. For example, an RZ23 disk at target ID 3 on bus 0 would be referred to as rz3; a TZK50 tape at target ID 5 on the second SCSI bus would be referred to as 13.

The SCSI bus has eight possible target device IDs. By default, one is allocated to the system. This allows for a maximum of seven target devices connected to a SCSI bus.

Restrictions

The ULTRIX SCSI device driver does not operate with optical disks, other than the Digital-supplied devices.

The SCSI driver attempts to support on a best effort basis, non-Digital-supplied winchester disks and magnetic tapes.

The following notes apply to the driver's handling of non-Digital-supplied disks:

- These disks are assigned a device type of RZxx, instead of RZ22, RZ23, RZ55, RZ56, or RZ57. The RZxx disks follow the same logical device naming scheme as the Digital-supplied disks.
- During the autoconfigure phase of the system startup, the driver prints the contents of the SCSI vendor ID, product ID, and the revision level fields of the inquiry data return by the SCSI device.
- RZxx disks are assigned a default partition table. The default table can be modified by editing the sz_rzxx_sizes[8] entry in the file `/usr/sys/data/scsi_data.c`. The `chpt` utility can also be used to

modify the partition table on a RZxx disk.

- The only logical unit number (LUN) supported for each target ID is 0.

See Also

rz(4), tz(4), chpt(8)

Name

sdc – RD31, RD32, RD53, RD54, RX33, RX23 Small VAX disk interface

Syntax

controller sdc0 at uba0 csr 0x200c0000 vector sdintr
disk rd0 at sdc0 drive 0
disk rd1 at sdc0 drive 1
disk rx2 at sdc0 drive 2

Description

This is a driver for the Digital Small VAX disk controller. This disk controller is used by both the workstation and multiuser configurations of the busless Small VAX processor. This controller also supports the RX23 floppy disk drive in the VAXstation 3100 model 30 processor.

The SDC driver uses the same disk format as the RQDX3 controller. Winchester disks formatted by the small VAX controller are compatible with RQDX3 formatted disks, but not with RQDX1 and RQDX2 formatted disks. The SDC driver implements dynamic bad block replacement in the same manner as the RQDX3 controller.

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 forth. The standard device names begin with rd, for Winchester disk drives 0 and 1 and rx, for the diskette drive 2 followed by the drive number and then a letter, a through h, for partitions 0 through 7.

The block files access the disk by the system's normal buffering mechanism and can 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. One read or write call results in one I/O operation, so raw I/O is considerably more efficient when many words are transmitted. The names of the raw files conventionally begin with an additional letter r, for example, `rrx@a`.

In raw I/O, counts should be a multiple of 512 bytes (a disk sector). In the same way, `seek` calls should specify a multiple of 512 bytes.

Disk Support

The SDC driver handles RD31, RD32, RD53, and RD54 Winchester disk drives on drives 0 and 1 and the RX33/RX23 floppy disk drive on drive 2. The RX33 drive reads and writes both single-density (400 Kbytes) RX50 floppy diskettes and double-density double-sided (1200 Kbytes) RX33 diskettes. The RX23 drive reads and writes high density (18 sectors per track) RX23 floppy diskettes and reads double density (9 sectors per track) RX23 floppy diskettes. Drive types are recognized in the autoconfiguration process. For constructing file systems, the partition sizes are required. Partition sizes are the same as those supported by the MSCP driver, `ra`(4). Partition sizes can be queried or changed by `chpt`(8).

The rd?a partition is usually used for the root file system, the rd?b partition as a paging area, and the rd?g partition for the usr file system.

Diagnostics

The following messages are printed at the console and written to the system error log file, `/usr/adm/syserr/syserr.`*<hostname>*.

sd%d:HARD_ERR: cannot read XBN
The driver cannot read the format information from the disk during autoconfiguration. The disk may not be formatted properly.

sd%d:HARD_ERR: CANNOT RECOVER FROM PREVIOUS BBR
Bad block replacement was interrupted in the middle when the system was last halted. The driver cannot successfully complete the bad block replacement. Try to recover all the data from the disk and reformat it. Refer to the *Guide to System Configuration File Maintenance* for additional information.

sd%d:HARD_ERR: Drive select failed
The driver cannot select the specified drive for doing I/O. Make sure the drive is on line.

sd%d:HARD_ERR: Invalid cylinder: %d
The driver tries I/O on a cylinder outside the valid range for the type of disk on the drive. This is a fatal error caused by the driver and should not happen.

sd%d:HARD_ERR: Invalid head:%d
The driver tries I/O on a head outside the valid range for the type of disk on the drive. This is a fatal error caused by the driver and should not happen.

sd%d:HARD_ERR: Forced Error Modifier set LBN %d
The forced error bit is set on the specified block. The block was found to be bad and has been replaced with a good block, but the data in the block is bad. Writing new data into this block will clear the forced error bit. Refer to the *Guide to System Configuration File Maintenance* for additional information.

sd%d:HARD_ERR: compare error
The driver received a compare error for the drive from the controller. This should be seen only for the floppy drive. For hard disks, the bad block will get replaced.

sd%d:HARD_ERR: eccerror
The driver received an ECC error for the drive from the controller.

sd%d:HARD_ERR: syncerr
The driver received a sync error for the drive from the controller. If it is the floppy drive, reinsert the floppy and repeat the command.

sd%d:HARD_ERR: bad sector
The controller detected the sector to be bad from the sector's ID field. This message should appear only for the floppy drive. For hard disks, the bad block will get replaced.

sd%d:HARD_ERR: WRITE FAULT
This is due to an internal error in the drive, such as an improper supply voltage. This message should appear only for hard disk drives.

The following messages are written to the system error file, `/usr/adm/syserr/syserr.`*<hostname>*, but not printed at the console..

sd:SOFT_ERR: stray interrupt
An unexpected interrupt was received (for example, when no I/O was pending). The interrupt is ignored.

sd:SOFT_ERR: No valid buffer
An interrupt was received when the driver was not ready to receive one. The interrupt is ignored. This should rarely happen.

sd%d:SOFT_ERR: Command not yet implemented thru interrupt, command = %c"
An interrupt was received for the command. This should not happen. The interrupt is ignored.

sd%d:SOFT_ERR: Unknown error type, UDC_CSTAT = %o, UDC_DSTAT = %o, DKC_STAT = %o"
The error type indicated by the controller for the last I/O is not any of the common ones. An I/O error is generated.

Files

```
/dev/rd[0-1][a-f]
/dev/rrd[0-1][a-f]"

/dev/rx2a"
/dev/rx2c"

/dev/rrx2a"
/dev/rrx2c"
```

See Also

dkio(4), nbuf(4), ra(4), rd(4), rx(4), chpt(8), uerf(8)

Name

sg – Small VAX color video subsystem

Syntax

device sg0 at uba0 csr 0x3c000000 flags 0x0f vector sgaint sgfint

Description

A small VAX color video subsystem provides a half-page or full-page, user-accessible bitmap display for graphics applications. The device consists of a 128-Kbyte block of dual port RAM, a VSXXX-AA mouse or VSXXX-AB tablet, and a 19-inch video monitor.

The subsystem device driver supports a hybrid terminal with three minor devices. The first device emulates a glass tty with a screen that appears as a 120-column by 80-row page that scrolls from the bottom. This device is capable of being configured as the system console.

The second minor device number is reserved for the mouse. This device is a source of mouse state changes. (A state change is defined as an X/Y axis mouse movement or button change.) When opened, the driver couples movements of the mouse with the cursor. Mouse position changes are filtered and translated into cursor position changes in an exponential manner. Rapid movements result in large cursor position changes. All cursor positions are range checked to ensure that the cursor remains on the display.

The third minor device is opened in the raw mode by default. Opening the third device makes the driver function like a pseudo-tty in that the output destined for the first minor device is channeled to the third instead.

If there is not a special cable (BCC08) on serial port 3 (printer port), the system will then use the color video as the system console. All input/output destined for `/dev/console` will use the color video instead. (This is done by overwriting the device switch tables.) There is a second set of device switch entries configured that may be used as an additional terminal, `tip/uucp` (hardwire, modem, or autodialer) connection, or user dial-up access. For further information, see `ss`(4).

The Hold Screen key is supported. The Small VAX color driver treats this key as if CTRL/S or CTRL/Q is typed. Pressing the Hold Screen key suspends the output if it is not already suspended. The output will be resumed by pressing this key again (if the output was suspended).

Files

```
/dev/console
/dev/sg0
/dev/sgscreen
```

See Also

ss(4), ttys(5), MAKEDEV(8)

sh(4)

Name

sh – Small VAX DHT32 serial line interface

Syntax

device sh0 at uba0 csr 0x38000000 flags 0xff vector shrint shxint

Description

A DHT32 provides eight asynchronous communication lines with no modem control.

Each line attached to the DHT32 communications multiplexer behaves as described in `tty`(4). Input and output for each line can be set independently to run at any of 16 speeds. See `tty`(4) for the coding.

Because no modem control is provided, a flags field of 0xff must be specified to ensure that all lines will be handled as data leads only.

Diagnostics

sh%d: DIAG. FAILURE
A failure has been detected by internal module diagnostics.

sh%d: recv. fifo overflow
The character input fifo overflowed before it could be serviced. This can happen if a hard error occurs when the CPU is running with elevated priority. Interrupts are disabled, and the system then prints a message on the console.

Files

`/dev/tty??`

See Also

tty(4), MAKEDEV(8)

Name

sm – Small VAX monochrome video subsystem

Syntax

device sm0 at uba0 csr 0x200f0000 flags 0x0f vector smvint

Description

A Small VAX monochrome video subsystem provides a half-page or full-page, user-accessible bitmap display for graphics applications. The device consists of a 128-Kbyte block of dual port RAM, a VSXXX-AA mouse or VSXXX-AB tablet, and a 19-inch video monitor.

The subsystem device driver supports a hybrid terminal with three minor devices. The first minor device emulates a glass tty with a screen that appears as a 120-column by 80-row page that scrolls from the bottom. This device is capable of being configured as the system console.

The second minor device is reserved for the mouse. This device is a source of mouse state changes. (A state change is defined as an X/Y axis mouse movement or button change.) When opened, the driver couples movements of the mouse with the cursor. Mouse position changes are filtered and translated into cursor position changes in an exponential manner. Rapid movements result in large cursor position changes. All cursor positions are range checked to ensure that the cursor remains on the display.

The third minor device provides an access path for console output that does not disturb the graphics display. The caller can open the device `/dev/smscreen`. When this device is open, the Small VAX monochrome driver redirects console device output to the input buffer of this device. This mechanism disables console output on the Small VAX monochrome screen and saves the output for later display. This preserves the graphic display integrity.

Input and output on the first and third minor devices are processed by the standard line disciplines.

If there is not a special cable (BCC08) on serial port 3 (printer port), and the optional video board is not present (or failed the self-test), the system will then use the monochrome video as the system console. All input/output destined for `/dev/console` will use the monochrome video instead. (This is done by overwriting the device switch tables.) There is a second set of device switch entries configured that may be used as an additional terminal, `tip/uucp` (hardwire, modem, or autodialer) connection, or user dial-up access. For further information, see `ss`(4).

The Hold Screen key is supported. The Small VAX monochrome driver treats this key as if CTRL/S or CTRL/Q is typed. Pressing the Hold Screen key suspends the output if it is not already suspended. The output will be resumed by pressing this key again (if the output was suspended).

Files

```
/dev/console
/dev/mouse
/dev/smscreen
```

See Also

ss(4), ttys(5), MAKEDEV(8)

Name

sp – Small VAX user supplied device driver interface

Syntax

device sp0 at uba0 csr 0x39000000 vector spintr

Description

The MicroVAX 2000 and MicroVAX 3100 system modules both have connectors that allow for the installation of a single option module. For example, the DHT32 is an option module (see `sh`(4)). The ULTRIX software includes drivers for the DHT32 and other supported options.

The `sp` driver is a set of routines that provides the linkages necessary to add a user-written device driver to the ULTRIX operating system. The `sp` files listed in the Files Section contain comments about interfacing a new driver to the ULTRIX operating system kernel.

Restrictions

The `sp` driver is not a guide to writing device drivers.

This driver can only be used with the MicroVAX 2000, MicroVAX 3100, VAXstation 2000, and the VAXstation 3100 systems. The usefulness of the `sp` driver on the VAXstation 2000 and VAXstation 3100 is further limited by option hardware mounting constraints on those systems.

Files

```
/sys/data/sp_data.c
/sys/io/uba/spreg.h
/sys/io/uba/sp.c
```

See Also

sh(4)

Name

ss – Small VAX serial line interface

Syntax

```
device ss0 at uba0 csr 0x200a0000 flags 0x0f
        vector ssrint ssxint
```

Description

The Small VAX serial line controller is similar to the DZQ11 4-line communications multiplexer. An `ss` interface provides four communication ports with partial modem control on port 2, adequate for dialup use. Only port 2 supports modem control (dialup access). All other ports must be operated as local lines. Each line attached to the serial line controller behaves as described in `tty`(4) and may be set to run at any of 16 speeds. For the encoding, see `tty`(4). However, configuration requirements dictate fixed speed operation of ports connected to the console terminal and graphics devices.

The Small VAX may be configured as a workstation or a multiuser timesharing system. For the workstation configuration, the `ss` ports are used as follows:

Port	Usage
0	Graphics device keyboard at 4800 BPS
1	Mouse or tablet at 4800 BPS
2	Communications (with modem control)/local terminal
3	Serial printer port

For the multiuser configuration, the `ss` port usage is:

Port	Usage
0	Console terminal at 9600 BPS
1	Local terminal line
2	Communications (with modem control)/local terminal
3	Local terminal or serial printer

For either configuration, a diagnostic console terminal may be connected to port 3 using a BCC08 cable. For the multiuser configuration, while the diagnostic console is connected, no other terminal devices can be connected. When the diagnostic console is in use, the processor may be halted by pressing the BREAK key.

The selection of which port to use for the console is made during the processor's power-on sequence and cannot be changed after power on. If the diagnostic console is connected, it is used; otherwise, the device connected to port 0 is the console.

For the `ss` device, the flags should always be specified as: flags 0x0f (all 4 lines hardwired). The state of port 2 can be established by specifying either modem or nomodem as part of the `/etc/ttys` file entry for tty02; see `ttys`(5). The default state of port 2 can be controlled by flags bit 2. Set 'flags 0x0f' for a hardwired line, 'flags 0x0b' for dialup operation (wait for carrier).

The `ss` driver operates in interrupt-per-character mode (all pending characters are flushed from the silo on each interrupt). Silo alarm mode is used by the DZQ11 driver at times of high input character traffic. This mode is not used by the `ss` driver, due to the need to track mouse or tablet position changes in real time.

VAXstation 3100 Communications and Printer Ports

The VAXstation 3100 has two MMJ (Modified Modular Jack) connectors located at the rear of the system box. These MMJ connectors allow connection of terminals, printers, and modems to the VAXstation 3100 system.

The ULTRIX logical names for these connectors are `/dev/tty02` (the MMJ closest to the power connector) and `/dev/tty03` (the MMJ next to the graphics connector).

Terminals and printers can be connected to either the `/dev/tty02` or `/dev/tty03` MMJ. Modems can be connected only to the `/dev/tty02` MMJ.

The VAXstation 3100 hardware provides only limited modem control support. The DTR (Data Terminal Ready) and DSR (Data Set Ready) signals are the only modem control signals available at the `/dev/tty02` MMJ. The ULTRIX device driver for `/dev/tty02` has been modified to allow modems to function with the limited modem control provided by the VAXstation 3100. For the modem to function properly, it must be configured to drop DSR when the carrier drops; that is, the modem should not continuously assert DSR.

Restrictions

The speed must be set to 9600 BPS on the console port and 4800 BPS on ports used by graphics devices. The console device must be set for 8-bit character length with one stop bit and no parity. The `ss` driver enforces these restrictions; that is, changing speeds with the `stty` command may not always work on these ports.

Diagnostics

ss0: input silo overflow
The 64-character input silo overflowed before it could be serviced. This can happen if a hard error occurs when the CPU is running with elevated priority, as the system then prints a message on the console with interrupts disabled.

Files

`/dev/console`	console terminal or graphics device
`/dev/tty00`	not used
`/dev/tty01`	local terminal – multiuser configuration only
`/dev/tty02`	dialup or local terminal
`/dev/tty03`	printer port or local terminal
`/dev/mouse`	mouse or tablet – workstation configuration only
`/dev/smscreen`	console message window for monochrome workstations
`/dev/sgscreen`	console message window for color workstations

See Also

console(4), devio(4), sm(4), tty(4), ttys(5), MAKEDEV(8)

Name

stc – TZK50 (VAXstation TK50) magnetic tape interface

Syntax

controller stc0 at uba0 csr 0x200c0080 vector stintr
tape st0 at stc0 drive 0

Description

The TZK50/TK50 combination provides a standard tape drive interface, as described in `mtio`(4). The TZK50 is supported only on the VAXstation 2000 and the MicroVAX 2000. This driver also supports n-buffered reads and writes to the raw tape interface (used with streaming tape drives). See `nbuf`(4) for further details.

Diagnostics

The following messages are printed at the console:

st0: arbitration failed after %d tries
After the indicated number of tries, the system gives up trying to arbitrate for the TK50 bus.

st0: device failed to select
The host was not able to select the TK50 device on the bus.

st0: device failed to reselect
The host was not able to reselect the TK50 device on the bus.

st0: parity error
A parity error was encountered as part of a command or status packet.

st0: bus reset
The TK50 bus has just been reset.

st0: buffer too large
A read or write was requested that exceeds the 16K maximum supported block size of this driver.

st0: aborting transfer
An error has occurred, and the request to the driver is being aborted.

st0: request sense data: %x %x %x %x %x %x %x %x %x %x
When an error occurs, the request sense data is the last command's status sent from the TZK50 controller.

st0: software error
The software has sent an illegal request to the drive.

st0: controller failed selftest
The TZK50 controller failed power on selftest. The drive is unuseable.

st0: drive failed selftest, code = 0x%x
The TZK50 tape drive failed power on selftest. The drive is unuseable. The drive error code is reported in hexadecimal.

st0: controller firmware revision %d not supported
The indicated controller firmware revision is not supported by ULTRIX.

st0: soft error, sense key = 0x%x
A nonfatal error has occurred. The sense key is part of the request sense data returned from the TZK50 controller.

st0: hard error, sense key = 0x%x
A fatal error has occurred. The sense key is part of the request sense data returned from the TZK50 controller.

Restrictions

The maximum block size supported by the stc driver is 16K bytes. Block sizes greater than 16,384 bytes produce the error message **st0: buffer too large.**

Files

```
/dev/rmt???
/dev/nrmt???
```

See Also

mtio(4), nbuf(4), MAKEDEV(8), uerf(8)

Name

tcp – Internet Transmission Control Protocol

Syntax

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

s = socket(AF_INET, SOCK_STREAM, 0);

Description

The TCP protocol provides reliable, flow-controlled, two-way transmission of data. It is a byte-stream protocol used to support the SOCK_STREAM abstraction. TCP uses the standard Internet address format and, in addition, provides a per-host collection of ''port addresses''. Thus, each address is composed of an Internet address specifying the host and network, with a specific TCP port on the host identifying the peer entity.

Sockets utilizing the TCP protocol are either ''active'' or ''passive''. Active sockets initiate connections to passive sockets. By default TCP sockets are created active; to create a passive socket the `listen`(2) system call must be used after binding the socket with the `bind`(2) system call. Only passive sockets can use the `accept`(2) call to accept incoming connections. Only active sockets can use the `connect`(2) call to initiate connections.

Passive sockets can ''underspecify'' their location to match incoming connection requests from multiple networks. This technique, termed ''wildcard addressing'', allows a single server to provide service to clients on multiple networks. To create a socket that listens on all networks, the Internet address INADDR_ANY must be bound. The TCP port can still be specified at this time. If the port is not specified, the system will assign one. Once a connection has been established, the socket's address is fixed by the peer entity's location. The address assigned the socket is the address associated with the network interface through which packets are being transmitted and received. Normally, this address corresponds to the peer entity's network.

TCP supports one socket option that is set with `setsockopt`(2) 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 that receive no replies, this packetization may cause significant delays. Therefore, TCP provides a Boolean option, TCP_NODELAY (from `<netinet/tcp.h>`, to defeat this algorithm. The option level for the `setsockopt` call is the protocol number for TCP, available from `getprotobyname`(3n).

Diagnostics

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

[EISCONN]	Try to establish a connection on a socket which already has one.

[ENOBUFS]	The system runs out of memory for an internal data structure.
[ETIMEDOUT]	A connection was dropped due to excessive retransmissions.
[ECONNRESET]	The remote peer forces the connection to be closed.
[ECONNREFUSED]	The remote peer actively refuses connection establishment (usually because no process is listening to the port).
[EADDRINUSE]	An attempt is made to create a socket with a port that has already been allocated.
[EADDRNOTAVAIL]	An attempt is made to create a socket with a network address for which no network interface exists.

See Also

getsockopt(2), socket(2), inet(4f), intro(4n), ip(4p)

Name

termio – System V terminal interface

Description

This section specificly describes the System V terminal interface. A general description of the available terminal interfaces is provided in `tty`(4).

When a terminal file is opened, it normally causes the process to wait until a connection is established. In practice, users' programs seldom open these files; they are opened by `getty` and become a user's standard input, output, and error files. The very first terminal file opened by the process group leader of a terminal file not already associated with a process group becomes the *control terminal* for that process group. The control terminal plays a special role in handling quit and interrupt signals, as discussed later. The control terminal is inherited by a child process during a `fork`(2). A process can break this association by changing its process group using `setpgrp`(2).

A terminal associated with one of these files ordinarily operates in full-duplex mode. Characters can be typed at any time, even while output is occurring. They 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. Currently, this limit is (MAX_INPUT) characters, as defined in `<limits.h>`. When the input limit is reached, all the saved characters are thrown away without notice.

Normally, terminal input is processed in units of lines. A line is delimited by a newline (ASCII LF) character, an end-of-file (ASCII EOT) character, or an end-of-line character. This means that a program attempting to read is suspended until an entire line has been typed. No matter how many characters are requested in the read call, at most one line will be returned. It is not, however, necessary to read a whole line at once; any number of characters can be requested in a read, even one, without losing information.

Some characters have special meaning when input. For example, during input *erase* and *kill,* processing is normally done. The erase character erases the last character typed, except that it will not erase beyond the beginning of a line. Typically the erase character is the number sign (#). The kill character kills (deletes) the entire input line, and optionally outputs a newline character. The default kill character is the at sign (@). Both characters operate on a key-stroke basis, independently of any backspacing or tabbing. Both the erase and kill characters can be entered literally by preceding them with the escape character (\). In this case the escape character is not read. The erase and kill characters can be changed.

Certain characters have special functions on input. These functions and their default character values are summarized as follows:

`INTR` (Rubout or ASCII DEL) generates an *interrupt* signal that is sent to all processes with the associated control terminal. Normally, each such process is forced to terminate, but arrangements can be made either to ignore the signal or to receive a trap to an agreed-upon location; see `signal`(3).

`QUIT` (CTRL/| or ASCII FS) generates a *quit* signal. Its treatment is identical to the interrupt signal except that, unless a receiving process has made other arrangements, it is not only terminated but a core image file (called **core**) is created in the current working directory.

`ERASE`	The number sign (#) erases the preceding character. It will not erase beyond the start of a line, as delimited by an NL, EOF, or EOL character.
`KILL`	The at sign (@) deletes the entire line, as delimited by an NL, EOF, or EOL character.
`EOF`	(CTRL/D or ASCII EOT) can be used to generate an end-of-file from a terminal. When received, all the characters waiting to be read are immediately passed to the program, without waiting for a 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 are passed back, which is the standard end-of-file indication.
`NL`	(ASCII LF) is the normal line delimiter. It can not be changed or escaped.
`EOL`	(ASCII NUL) is an additional line deliminter, like NL. It is not normally used.
`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. The START/STOP characters can not be changed or escaped.
`MIN`	Used to control terminal I/O when the ICANON flag is not set in the c_lflag. Input processing behaves as described in the **MIN/TIME Interaction** section that follows.
`TIME`	Used to control terminal I/O when the ICANON flag is not set in the c_lflag. Input processing behaves as described in the **MIN/TIME Interaction** section that follows.

The character values for INTR, QUIT, ERASE, KILL, EOF, MIN, TIME, and EOL can be changed to suit individual tastes. The ERASE, KILL, and EOF characters can be escaped by a preceding backslash (\) character, in which case no special function is performed.

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 by putting them in the output queue as they arrive. If a process produces characters more rapidly than they can be typed, it is suspended when its output queue exceeds some limit. When the queue has drained down to some threshold, the program is resumed.

When the carrier signal from the data-set drops, a *hang-up* signal, SIGHUP, is sent to all processes that have this terminal as the control terminal. Unless other arrangements have been made, this signal causes the processes to terminate. If the hang-up signal is ignored, any subsequent read returns with an end-of-file indication. Thus, programs that read a terminal and test for end-of-file can terminate appropriately when hung up on.

Several `ioctl`(2) system calls apply to terminal files. The primary calls use the following structure, defined in <`termio.h`>:

```
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 initial values for each function are as follows:

VINTR	DEL
VQUIT	FS
VERASE	#
VKILL	@
VEOF	EOT
VEOL	NUL
VMIN	6
VTIME	1

The *c_iflag* field describes the basic terminal input control:

IGNBRK	Ignore break condition.
BRKINT	Signal interrupt on break.
IGNPAR	Ignore characters with parity errors.
PARMRK	Mark parity errors.
INPCK	Enable input parity check.
ISTRIP	Strip character.
INLCR	Map NL to CR on input.
IGNCR	Ignore CR.
ICRNL	Map CR to NL on input.
IUCLC	Map uppercase to lowercase on input.
IXON	Enable start/stop output control.
IXANY	Enable any character to restart output.
IXOFF	Enable start/stop input control.

If IGNBRK is set, the break condition (a character framing error, with data all zeros) is ignored, that is, not put on the input queue and therefore not read by any process. Otherwise, if BRKINT is set, the break condition generates an interrupt signal and flushes both the input and output queues. If IGNPAR is set, characters with other framing and parity errors are ignored.

If PARMRK is set, a character with a framing or parity error that is not ignored is read as the three-character sequence: 0377, 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 0377 is read as 0377, 0377. If PARMRK is not set, a framing or parity error that is not ignored is read as the character NUL (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 seven bits. Otherwise, all eight 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 uppercase letter is translated into the corresponding lowercase letter.

If IXON is set, start/stop output control is enabled. A received STOP character suspends output and a received START character restarts output. All start/stop characters are ignored and not read. If IXANY is set, any input character restarts output that has been suspended.

If IXOFF is set, the system transmits START/STOP characters when the input queue is nearly empty/full.

The initial input control value is all-bits-clear.

The *c_oflag* field specifies the system treatment of output:

OPOST	Postprocess output.
OLCUC	Map lower case to upper on output.
ONLCR	Map NL to CR-NL on output.
OCRNL	Map CR to NL on output.
ONOCR	No CR output at column 0.
ONLRET	NL performs CR function.
OFILL	Use fill characters for delay.
OFDEL	Fill is DEL or else NUL.
NLDLY	Select newline delays:
NL0	Newline delay type 0.
NL1	Newline delay type 1.
CRDLY	Select carriage-return delays:
CR0	Carriage-return delay type 0.
CR1	Carriage-return delay type 1.
CR2	Carriage-return delay type 2.
CR3	Carriage-return delay type 3.
TABDLY	Select horizontal-tab delays:
TAB0	Horizontal-tab delay type 0.
TAB1	Horizontal-tab delay type 1.
TAB2	Horizontal-tab delay type 2.
TAB3	Expand tabs to spaces.
BSDLY	Select backspace delays:
BS0	Backspace delay type 0.
BS1	Backspace delay type 1.
VTDLY	Select vertical-tab delays:
VT0	Vertical-tab delay type 0.
VT1	Vertical-tab delay type 1.
FFDLY	Select form-feed delays:

FF0	Form-feed delay type 0.
FF1	Form-feed delay type 1.

If OPOST is set, output characters are post-processed as indicated by the remaining flags, otherwise characters are transmitted without change.

If OLCUC is set, a lowercase letter is transmitted as the corresponding uppercase letter. This function is often used in conjunction with IUCLCS.

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 is set to 0 and the delays specified for CR are used. Otherwise, the NL character is assumed to do just the line-feed function; the column pointer remains 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 is 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 it is NUL.

If a form-feed or vertical-tab delay is specified, it lasts for about two seconds.

A newline delay lasts about 0.10 seconds. If ONLRET is set, the carriage-return delays are used instead of the newline delays. If OFILL is set, two fill characters is transmitted.

A carriage-return delay type 1 is dependent on the current column position. The type 2 delay is about 0.10 seconds; the type 3 delay is about 0.15 seconds. If OFILL is set, delay type 1 transmits two fill characters; type 2 transmits four fill characters.

A horizontal-tab delay type 1 is dependent on the current column position. The type 2 delay is about 0.10 seconds. Type 3 specifies that tabs are to be expanded into spaces. If OFILL is set, two fill characters is transmitted for any delay.

A backspace delay lasts about 0.05 seconds. If SM 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 all bits clear.

The *c_cflag* field describes the hardware control of the terminal:

CBAUD	Baud rate:
B0	Hang up
B50	50 baud
B75	75 baud
B110	110 baud
B134	134.5 baud
B150	150 baud
B200	200 baud
B300	300 baud
B600	600 baud
B1200	1200 baud
B1800	1800 baud

B2400	2400 baud
B4800	4800 baud
B9600	9600 baud
B19200	19200 baud
B38400	38400 baud
EXTA	External A (Same as B19200)
EXTB	External B (Same as B38400)

CSIZE	Character size:
CS5	5 bits
CS6	6 bits
CS7	7 bits
CS8	8 bits
CSTOPB	Send two stop bits, otherwise one.
CREAD	Enable receiver.
PARENB	Parity enable.
PARODD	Odd parity, otherwise even.
HUPCL	Hang up on last close.
CLOCAL	Local line, otherwise dial-up.

The CBAUD bits specify the baud rate. The zero baud rate, B0, is used to hang up the connection. If B0 is specified, the data-terminal-ready signal will not be asserted. Normally, this will disconnect the line. 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, only one stop bit is used. For example, at 110 baud, two stops 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 line will be disconnected when the last process with the line open closes it or terminates. That is, the data-terminal-ready signal will not be asserted.

If CLOCAL is set, the line is assumed to be a local, direct connection with no modem control. Otherwise, modem control is assumed.

The initial hardware control value after open is B300, CS8, CREAD, HUPCL.

The *c_lflag* field of the argument structure is used by the line discipline to control terminal functions. The basic line discipline (0) provides the following:

ISIG	Enable signals.
ICANON	Canonical input (erase and kill processing).
XCASE	Canonical upper/lower presentation.
ECHO	Enable echo.
ECHOE	Echo erase character as BS-SP-BS.
ECHOK	Echo NL after kill character.
ECHONL	Echo NL.
NOFLSH	Disable flush after interrupt or quit.

If ISIG is set, each input character is checked against the special control characters INTR, SWTCH, and QUIT. 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. These functions can be disabled individually by changing the value of the control character to an unlikely or impossible value (for example, 0).

If ICANON is set, canonical processing is enabled. This enables the erase and kill edit functions, and the assembly of input characters into lines delimited by NL, EOF, and EOL. If ICANON is not set, read requests are satisfied directly from the input queue. A read is not 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 MIN and TIME values are stored in the position for the EOF and EOL characters, respectively. The time value represents tenths of seconds.

If XCASE is set and if ICANON is set, an upper-case letter is accepted on input by preceding it with a backslash (\), and is output preceded by a backslash (\). 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.

When ICANON is set, the following echo functions are possible. If ECHO and ECHOE are set, the erase character is echoed as ASCII BS SP BS, which clears the last character from a CRT screen. If ECHOE is set and ECHO is not set, the erase character is echoed as ASCII SP BS. If ECHOK is set, the NL character is echoed after the kill character to emphasize that the line is deleted. Note that an escape character preceding the erase or kill character removes any special function. If ECHONL is set, the NL character is echoed, even if ECHO is not set. This is useful for terminals set to local echo (so-called half duplex). Unless escaped, the EOF character is not echoed. Because EOT is the default EOF character, terminals that respond to EOT are prevented from hanging up.

If NOFLSH is set, the normal flush of the input and output queues associated with the quit, switch, and interrupt characters will not be done.

The initial line-discipline control value is all bits clear.

The *c_line* field of the argument structure is used to specify the line discipline. Support is provided for the basic termio line discipline only. For this reason, the value of this field is irrelevant and should be set to zero (0) by convention.

The primary `ioctl`(2) system calls have the form:

ioctl (fildes, command, arg)
struct termio *arg;

The commands using this form are:

TCGETA	Get the parameters associated with the terminal and store in the *termio* structure referenced by *arg*.
TCSETA	Set the parameters associated with the terminal from the structure referenced by *arg*. The change is immediate.

TCSETAW	Wait for the output to drain before setting the new parameters. This form should be used when changing parameters that will affect output.
TCSETAF	Wait for the output to drain, then flush the input queue and set the new parameters.

Additional `ioctl`(2) calls have the form:

ioctl (fildes, command, arg)
int arg;

The commands using this form are:

TCSBRK	Wait for the output to drain. If *arg* is 0, send a break (zero bits for 0.25 seconds).
TCXONC	Start/stop control. If *arg* is 0, suspend output; if 1, restart suspended output.
TCFLSH	If *arg* is 0, flush the input queue; if 1, flush the output queue; if 2, flush both the input and output queues.

MIN/TIME Interaction

MIN represents the minimum number of characters that should be received when the read is satisfied (that is, the characters are returned to the user). TIME is a timer of 0.10 second granularity used to time-out bursty and short-term data transmissions. The four possible values for MIN and TIME and their interactions follow:

1. MIN > 0, TIME > 0. In this case, TIME serves as an intercharacter timer activated after the first character is received and reset upon receipt of each character. MIN and TIME interact as follows:

As soon as one character is received the intercharacter timer is started.

If MIN characters are received before the intercharacter timer expires, the read is satisfied.

If the timer expires before MIN characters are received, the characters received to that point are returned to the user.

A `read`(2) operation will sleep until the MIN and TIME mechanism are activated by the receipt of the first character; thus, at least one character must be returned.

2. MIN > 0, TIME = 0. In this case, because TIME = 0, the timer plays no role and only MIN is significant. A `read`(2) operation is not satisfied until MIN characters are received.

3. MIN = 0, TIME > 0. In this case, because MIN = 0, TIME no longer serves as an intercharacter timer, but now serves as a read timer that is activated as soon as the `read`(2) operation is processed. A `read`(2) operation is satisfied as soon as a single character is received or the timer expires, in which case, the `read`(2) operation would not return any characters.

4. MIN = 0, TIME = 0. In this case, return is immediate. If characters are present, they are returned to the user.

Name

termios – POSIX terminal interface

Description

Interface Characteristics

The POSIX terminal interface is provided to control asynchronous communications ports, pseudoterminals, and the special file, `/dev/tty`. The ULTRIX operating system also provides a SVID termio terminal interface as defined in `termio`(4) and a Berkeley terminal interface as defined in `tty`(4). The following sections describe the general terminal interface as defined by the POSIX operating system specification. For a general overview of the various terminal interfaces, refer to the subsection entitled Terminal interface definitions in `tty`(4).

The POSIX termios specification defines a set of terminal-related functions and attributes that facilitate the development of portable programs. The specification allows for local extensions to the terminal interface. Throughout this description, all local extensions to the termios interface are noted. Programs that are written to be highly portable should avoid the usage of local extensions.

Opening a Terminal Device File

When a terminal file is opened, it normally causes the process to wait until a connection is established. In practice, application programs seldom open these files; they are opened by special programs such as `getty`(8) and become a user's standard input, output, and error files.

As described in `open`(2), opening a terminal device file with the O_NONBLOCK flag clear causes the process to block until the terminal device is ready and available. If CLOCAL mode is not set, blocking occurs until a connection is established. If CLOCAL mode is set, or the O_NONBLOCK flag is specified in the `open`(2), the `open`(2) returns a file descriptor, without waiting for a connection to be established.

Process Groups

A terminal may have a foreground process group associated with it. This foreground process group plays a special role in handling signal-generating input characters, as discussed in the Special Characters description.

A command interpreter that is capable of supporting job control, `csh`(1) for example, 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 foreground process group can 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 foreground process group. See the Job Access Control description for more information.

The Controlling Terminal

A terminal can belong to a process as its controlling terminal. Each process of a session that has a controlling terminal has the same controlling terminal. A terminal may be the controlling terminal for at most one session. If a session leader has no controlling terminal and opens a terminal device file that is not already associated with a session, without using the O_NOCTTY `open`(2) flag, the terminal then becomes the controlling terminal of the session leader. If a process that is not a session leader opens a terminal file, or the O_NOCTTY option is used with `open`(2), that terminal cannot become the controlling terminal of the calling process. When a controlling terminal becomes associated with a session, its foreground process group is set to the process group of the session leader.

The controlling terminal is inherited by a child process during a `fork`(2) function. A process relinquishes its controlling terminal when it creates a new session with the `setsid`(2) function, or when all file descriptors associated with the controlling terminal have been closed.

When a controlling process terminates, the controlling terminal is disassociated from the current session, allowing the terminal to be acquired as a controlling terminal by a new session process group leader. Subsequent access to the terminal by other processes in the earlier session may be denied, with attempts to access the terminal treated as if a modem disconnect had been sensed.

Closing a Terminal Device File

When the last process that has the terminal line open closes the terminal line, the process waits for all output to clear and any input is discarded. The wait does not exceed four minutes, preventing the line from becoming hung if a progress is not made in the final output. After the final output has been transmitted, any pending input is flushed. If HUPCL is set in the control modes, and the communications port supports a disconnect function, the terminal device performs a disconnect.

Modem Disconnect

When a modem disconnect is detected by the terminal interface for a controlling terminal, and the CLOCAL bit is not set in the control flag, the SIGHUP signal is sent to all the controlling processes associated with the terminal. Unless other arrangements have been made, this signal causes the controlling processes to terminate. If SIGHUP is ignored or caught, any subsequent read returns with an end-of-file indication. Thus, programs that read a terminal file and test for end-of-file can terminate appropriately after a disconnect. Any subsequent `write`(2) to the terminal device returns -1 with *errno* set to [EIO], until the device is closed.

Terminal Access Control

If a process is in the foreground process group (nonzero) of its controlling terminal (that is, if the process is a foreground process), `read`(2) operations are allowed as described in the Input Processing and Reading Characters description. Any attempts by a process in a background process group to read from its controlling terminal causes its process group to receive a SIGTTIN signal, unless one of the following is true:

- If the reading process is ignoring or blocking the SIGTTIN signal.
- If the process group of the reading process is orphaned, the `read`(2) returns -1, with *errno* set to [EIO], and no signal is sent.

Attempts by a process in a background process group to write to its controlling terminal causes the process group to receive a SIGTTOU signal, unless one of the following is true:

- If TOSTOP is not set, or if TOSTOP is set and the process is ignoring or blocking the SIGTTOU signal, the process is allowed to write to the terminal and the SIGTTOU is not sent.

- If TOSTOP is set, and the process group of the writing process is orphaned, and the writing process is not blocking SIGTTOU, the `write`(2) returns -1, with *errno* set to [EIO], and no signal is sent.

Certain calls that set terminal parameters are treated in the same fashion as `write`(2), except that TOSTOP is ignored. That is, the effect is identical to that of terminal writes when TOSTOP is set. See the Control Functions description for more information.

Input Processing and Reading Characters

A terminal device associated with a terminal device file operates in full-duplex mode, so that characters can arrive while output is occurring. Each terminal device file has associated with it an input queue, in which incoming characters are stored by the system before being read by a process. The system imposes a limit (MAX_INPUT), on the number of bytes that can be stored in the input queue. MAX_INPUT is limited to 256 characters. When ICANON is not set and the system's character input buffers become full, the input buffer is flushed without notice. This causes all the characters in the input queue to be lost. If ICANON is set and the system's character input buffers become full, the driver discards additional characters and echos a bell (ASCII BEL) to notify the user of the full condition.

Depending on whether or not the terminal device is in canonical mode or noncanonical mode, two general types of input processing are available. See the Canonical Mode Input Processing and Noncanonical Mode Input Processing descriptions for more information. Additionally, input characters are processed according to the c_iflag and c_lflag fields. See the Input Modes and Local Modes descriptions. Such processing can include echoing, which, in general, means transmitting input characters immediately back to the terminal when they are received from the terminal. This is useful for terminals that can operate in full-duplex mode. The manner in which characters are provided to a process reading from a terminal device is dependent on whether the terminal file is in canonical or noncanonical mode.

Another dependency is whether or not the O_NONBLOCK is set by `open`(2) or `fcntl`(2). If the O_NONBLOCK flag is clear, then the read request is blocked until data is available or a signal has been received. If the O_NONBLOCK flag is set, then the read request completes, without blocking, in one of the following ways:

- If there is enough data available to satisfy the entire request, the read completes successfully and returns the number of bytes read.
- If there is not enough data available to satisfy the entire request, the read completes successfully, having read as much data as possible and returns the number of bytes it was able to read.
- If data is not available, the read returns a –1, with *errno* set to EAGAIN.

As stated previously, the availability of data is dependent on the input processing mode. The input processing mode can be either canonical or non-canonical. The following sections discuss these modes in detail.

Canonical Mode Input Processing

In canonical mode input processing, terminal input is processed in units of lines. A line is delimited by a new-line (\n) character (ASCII LF), an end-of-file character (ASCII EOF), or a user-defined end-of-line character (EOL) . See the description of Special Characters for more information on EOF and EOL.

A read request cannot be satisfied until an entire line has been typed or a signal has been received. Regardless of the number of characters requested in the read call, at most one line is returned. However, it is not necessary to read a whole line at once; one or more characters can be requested in a read without losing information.

MAX_CANON (256) defines the maximum number of input characters the system can buffer in canonical mode. When this limit is exceeded, the system discards additional input.

Erase and kill processing occur when either the ERASE or KILL character is received. This processing affects data in the input queue that has not been delimited by a new-line (NL), end-of-file (EOF), or end-of-line (EOL) character. The data that is not delimited creates the current line. The ERASE character deletes the last character in the current line, if there is any. The KILL character deletes all data in the current line, if there is any. The ERASE and KILL characters have no effect if there is no data in the current line. The ERASE and KILL characters themselves are not placed in the input queue.

The reprint (RPRNT) character retypes pending input beginning on a new line. Retyping is automatic, if characters that would normally be erased from the screen are interspersed with program output.

Noncanonical Mode Input Processing

In noncanonical mode input processing, input characters are not assembled into lines, and erase and kill processing do not occur. The values of the special characters MIN and TIME are used to determine how to process the characters received. MIN and TIME are defined by the c_cc array of special control characters.

MIN represents the minimum number of characters that should be received when the read is satisfied (for example, the characters are returned to the user). TIME is a timer of 0.1 second granularity that is used to time out bursty and short term data transmissions. If MIN is greater than MAX_INPUT (256), the value of MIN is truncated to be MAX_INPUT. The four possible values for MIN and TIME and examples of their interactions are as follows:

1. MIN > 0, TIME > 0.
In this case, TIME serves as an intercharacter timer and is activated after the first character is received. Because it is an intercharacter timer, it is reset after a character is received. The interaction between MIN and TIME provokes the intercharacter time to start as soon as one character is received. If MIN characters are received before the intercharacter timer expires, the read is satisfied.

If the timer expires before MIN characters are received, the characters received to that point are returned to the user. If TIME expires, at least one character is returned, because the timer would not have been enabled unless a character was received. The read blocks until the MIN and TIME mechanisms are activated by the receipt of the first character, or a signal is received.

2. MIN > 0, TIME = 0.
In this case, the value of TIME is zero, the timer is inactive. However, MIN is significant. A pending read is not satisfied until MIN characters or a signal are received. That is, the pending read sleeps until MIN characters are received. A program that uses this example to read record-based terminal I/O can indefinitely block the read operation.

3. MIN = 0, TIME > 0.
In this case, because MIN = 0, TIME does not represent an intercharacter timer. Instead, it serves as a read timer that is activated as soon as the `read`(2) function is processed. A read is satisfied as soon as a single character is received or the read timer expires. Note, if the timer expires, a character is not returned. If the timer does not expire, a read is only satisfied if a character is received. For example, the read cannot block indefinitely waiting for a character. If a character is not received within TIME*0.1 seconds after the read is initiated, the read returns a value of zero, having read no data.

4. MIN = 0, TIME = 0.
In this case, only the minimum number of characters requested or the number of characters currently available are returned without waiting for more characters to be input. In this example, the return is immediate.

The following list summarizes the previous examples:

- The interactions of MIN and TIME are not symmetric. For example, when MIN > 0 and TIME = 0, TIME has no effect. However, if MIN = 0 and TIME > 0, MIN is activated by the receipt of a single character.
- When MIN > 0 and TIME > 0, TIME represents an intercharacter timer; when MIN = 0, TIME > 0, TIME represents a read timer.

The previous summary highlights the dual purpose of the MIN and TIME feature. Cases 1 and 2 handle burst mode activity (such as file transfer programs) where a program would like to process at least MIN characters at a time. In case 1, the intercharacter timer is activated by a user as a safety precaution. However, in case 2, it is turned off.

Cases 3 and 4 exist to handle single character timed transfers. These examples 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 3, the read is timed. However, in case 4, it is not.

Note that MIN is always a minimum. It does not denote a record length. That is, if a program performs a read of 20 bytes, MIN has a value of 10, and there are 25 characters present, 20 characters are returned to the user.

Writing Characters and Output Processing

When a process writes one or more characters to a terminal device file, they are processed according to the c_oflag (see Output Modes). The terminal interface provides a buffering mechanism. For example, when a call to `write`(2) completes, all of the characters written have been scheduled for transmission to the device, but the transmission is not necessarily complete. The characters are transmitted to the terminal as soon as previously written characters have output successfully.

Signal Handling

Signals caught during a `read`(2), `write`(2), or other operation on the file descriptor associated with the terminal file are handled appropriately, as described in `signal`(3).

Special Characters

Certain characters have special functions on input or output. These functions are:

INTR	Special character on input that is recognized if the ISIG flag is enabled. Generates a SIGINT signal that is sent to all processes in the foreground process group associated with the terminal. If ISIG is set, the INTR character is discarded when processed. The default value is octal 0177.
QUIT	Special character on input that is recognized if the ISIG flag is enabled. Generates a SIGQUIT signal that is sent to all processes in the foreground process group associated with the terminal. If ISIG is set, the QUIT character is discarded when processed. The default value is CTRL/| (ASCII FS).
ERASE	Special character on input that is recognized if the ICANON flag is set. Erases the last character in the current line. It cannot erase beyond the start of a line, as delimited by an NL, EOF, or EOL character. If ICANON is set, the ERASE character is discarded when processed. The default value is the number sign (#).
KILL	Special character on input that is recognized if the ICANON flag is set. Deletes the entire line, as delimited by an NL, EOF, or EOL character. If ICANON is set, the KILL character is discarded when processed. The default value is the at sign (@).
EOF	Special character on input that is recognized if the ICANON flag is set. This character is used to generate an end of file (EOF) from the 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 (that is, if the EOF occurred at the beginning of a line), a byte count of zero is returned from `read`(2), representing an end-of-file indication. If ICANON is set, the EOF character is discarded when processed. The default value is CTRL/D (ASCII EOT).
NL	Special character on input that is recognized if the ICANON flag is set. It is the assigned line delimiter (\n). It cannot be changed.
EOL	Special character on input that is recognized if the ICANON flag is set. It is an additional line delimiter, like NL. The default value is POSIX_V_DISABLE, which is used to specify that this special character is ordinarily not used.
SUSP	Special character on input that is recognized if the ISIG flag is enabled. Generates a SIGTSTP signal that is sent to all processes in the foreground process group associated with the terminal. This signal is used by the job control code to change from the current job to the controlling job. If ISIG is set, the SUSP character is discarded when processed. The default value is CTRL/Z (ASCII SUB).

STOP	Special character on both input and output that is recognized if the IXON (input) or IXOFF (output) flag is set. 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. If IXON is set, the STOP character is discarded when processed. The default value is CTRL/S (ASCII DC3).
START	Special character on both input and output that is recognized if the IXON (input) or IXOFF (output) flag is set. Can be used to resume output that has been suspended by a STOP character. While output is not suspended, START characters are ignored and not read. If IXON is set, the STOP character is discarded when processed. The default value is CTRL/Q (ASCII DC1).
CR	Special character on input that is recognized if the ICANON flag is set. The value is (\r) and this value is not changeable. When ICANON and ICRNL are set and IGNCR is not set, this character is translated into an NL, and it has the same effect as an NL character.
DSUSP	Special character on input used as a delayed suspend character. DSUSP is only recognized if the IEXTEN flag is set. Similar to the SUSP special character, a SIGTSTP signal is issued. The process cannot actually stop until the next character is input. If IEXTEN is set, the DSUSP character is discarded when processed. The default value is CTRL/Y (ASCII EM).
RPRNT	Special character on input used to force the present input line to be re-echoed to the terminal after a newline character. RPRNT is only recognized if the IEXTEN flag is set. If IEXTEN is set, the RPRNT character is discarded when processed. The default value is CTRL/R (ASCII DC2).
FLUSH	Special character on input that causes output to the terminal to be flushed until another flush character is typed or more input is received on the terminal. FLUSH is recognized only if the IEXTEN flag is set. If IEXTEN is set, the FLUSH character is discarded when processed. The default value is CTRL/O (ASCII SI).
WERASE	Special character on input used to erase the preceding word of the input queue. The word erase operation erases characters up to (and not including) a TAB, NL, space, or backslash (\) character. Word erase cannot erase beyond the start of a line as delimited by NL, EOF, or EOL. WERASE is only recognized if the IEXTEN flag is set. If IEXTEN is set, the WERASE character is discarded when processed. The default value is CTRL/W (ASCII ETB).
LNEXT	Special character on input used to disassociate any special meaning that the next input character has. This allows for the input of characters that would otherwise be interpreted as special characters. LNEXT is only recognized if the IEXTEN flag is set. If IEXTEN is set, the LNEXT character is discarded when processed. The default value is CTRL/V (ASCII SYN).
QUOTE	Special character on input used to enter a literal ERASE or KILL character. The same functionality could be achieved through the use of

the LNEXT character, but QUOTE is included for backward compatibility. The default value is a backslash (\). QUOTE is only recognized if the IEXTEN flag is set. If IEXTEN is set, the QUOTE character is discarded when processed.

The values for INTR, QUIT, ERASE, KILL, EOF, EOL, SUSP, START, STOP, DSUSP, RPRNT, FLUSH WERASE, LNEXT, and QUOTE are changeable to suit individual tastes. The following special characters are local extensions of DSUSP, RPRNT, FLUSH, WERASE, LNEXT, and QUOTE.

Special character functions can be disabled individually by setting them to the constant POSIX_V_DISABLE, which is defined to be zero. The POSIX_V_DISABLE character is always read if received, and never causes a special character function. With the exception of NL and EOL, the special characters cannot be passed up to the reading process.

If two or more special characters have the same value, the function performed when that character is received is undefined. More than one special character can be set to POSIX_V_DISABLE to disable the control function normally associated with the special character.

Settable Parameters

Routines that need to control certain terminal I/O characteristics can do so by using the termios structure as defined in the header `<termios.h>`. The members of this structure include:

```
        Member        Member      Description
        Type          Name

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    */
       cc_t          c_cc[NCCS] /* control chars */
       cc_t          c_line;     /* line discipline */
}
```

The types tcflag_t and cc_t are defined in the header `<termios.h>`.

Input Modes

The c_iflag field describes the basic terminal input control:

Mask Name	Description
IGNBRK	Ignore the break condition.
BRKINT	Signal interrupt on break.
IGNPAR	Ignore characters with parity errors.
PARMRK	Mark parity errors.
INPCK	Enable input parity check.
ISTRIP	Strip character.
INLCR	Map NL to CR on input.
IGNCR	Ignore CR.

ICRNL	Map CR to NL on input.
IXON	Enable start/stop output control.
IXOFF	Enable start/stop input control.

A break condition is defined as a sequence of zero-valued bits that continues for more than the time to send one byte. The entire sequence of zero-valued bits is interpreted as a single break condition, even if it continues for a time equivalent to more than one byte.

If IGNBRK is set, a break condition 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, the break condition generates a single SIGINT signal and flushes both the input and output queues. If neither IGNBRK or BRKINT is set, a break condition is read as a single \0 (ASCII NUL), or, if PARMRK is set, as \377,\0,\0.

If IGNPAR is set, a byte with a framing or parity error (other than break) is ignored.

If PARMRK is set and IGNPAR is not set, a character with a framing or parity error (other than break) that is not ignored is given to the application as the 3-character sequence \377,\0,X, where \377, \0 is a 2-character flag preceding each sequence and 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 given to the application as \377, \377. If either IGNPAR or PARMRK is set, a framing or parity error (other than break) that is not ignored is given to the application as a single character \0.

If INPCK is set, input parity checking is enabled. If INPCK is not set, input parity checking is disabled, allowing output parity generation without input parity errors. Parity checking can be enabled, even if parity checking is not enabled. If parity detection is enabled, but input parity checking is disabled, the hardware that connects to the terminal recognizes the parity bit, but the terminal special file does not check whether this bit is set correctly or not.

If ISTRIP is set, valid input characters are first stripped to seven bits; otherwise, all eight bits are processed.

If INLCR is set, a received NL character is translated into a CR character. If IGNCR is set, a received CR character is ignored (not read). If IGNCR is not set and ICRNL is set, a received CR character is translated into a NL character.

If IXON is set, start/stop output control is enabled. A received STOP character suspends output and a received START character restarts output. When IXON is set, START and STOP characters are not read, but merely perform flow control functions. When IXON is not set, the START and STOP characters are read.

If IXOFF is set, start/stop input control is enabled. The system transmits STOP characters which cause the terminal device to stop transmitting data when the number of characters in the input queue attempt to exceed MAX_INPUT (256). As soon as the device can continue transmitting data without risk of an overflow, START characters are transmitted which cause the terminal device to resume transmitting data.

The initial input control value after `open`(2) is 0 (all settings off).

Output Modes

The c_oflag field specifies the terminal interface's treatment of output. Because OPOST is the only output flag defined by the POSIX standard, all of the other definitions are local extensions to the standard.

Mask Name	Description
OPOST	Postprocess output.
OLCUC	Map lower case to upper on output.
ONLCR	Map NL to CR-NL on output.
OCRNL	Map CR to NL on output.
ONOCR	No CR output at column 0.
ONLRET	NL performs CR function.
OFILL	Use fill characters for delay.
OFDEL	Fill is DEL, else NUL.
NLDLY	Select new-line delays:
NL0	New-line delay type 0.
NL1	New-line delay type 1.
CRDLY	Select carriage-return delays:
CR0	Carriage-return delay type 0.
CR1	Carriage-return delay type 1.
CR2	Carriage-return delay type 2.
CR3	Carriage-return delay type 3.
TABDLY	Select horizontal-tab delays:
TAB0	Horizontal-tab delay type 0.
TAB1	Horizontal-tab delay type 1.
TAB2	Horizontal-tab delay type 2.
TAB3	Expand tabs to spaces.
BSDLY	Select backspace delays:
BS0	Backspace delay type 0.
BS1	Backspace delay type 1.
VTDLY	Select vertical-tab delays:
VT0	Vertical-tab delay type 0.
VT1	Vertical-tab delay type 1.
FFDLY	Select form-feed delays:
FF0	Form-feed delay type 0.
FF1	Form-feed delay type 1.

If OPOST is set, output characters are post-processed as indicated by the remaining flags. Otherwise, characters are transmitted without change.

If OLCUC is set, a lowercase letter is transmitted as the corresponding uppercase letter. 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 is set to 0, and the delays specified for CR are used. Otherwise the NL character is assumed to do just the line-feed function; the column pointer remains unchanged. The column pointer is also set to 0 if the CR character is actually transmitted.

Delay bits specify transmission stops that allow 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 are transmitted for delay instead of a timed delay. This is useful for high baud rate terminals which need only a minimal delay. If OFDEL is set, the fill character is DEL. Otherwise, it is NUL.

The delay specifications for NLDLY, CRDLY, TABDLY, BSDLY, VTDLY and FFDLY are meant to serve as masks for the respective delay field.

If a form-feed or vertical-tab delay is specified, either delay lasts approximately two seconds.

New-line delay lasts about 0.10 seconds. If ONLRET is set, the carriage-return delays are used instead of the new-line delays. If OFILL is set, two fill characters are transmitted.

Carriage-return delay for type 1 depends on the current column position, type 2 is approximately 0.10 seconds, and type 3 is approximately 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 depends on the current column position. Type 2 is approximately 0.10 seconds. Type 3 specifies that tabs are to be expanded into spaces. If OFILL is set, two fill characters are transmitted for any delay.

Backspace delay lasts about 0.05 seconds. If OFILL is set, one fill character is transmitted.

The actual delays depend on line speed and system load.

The initial output control value after open(2) is 0 (all settings off).

Control Modes

The c_cflag field describes the hardware control of the terminal:

Mask Name	Description
CSIZE	Character size:
CS5	5 bits
CS6	6 bits
CS7	7 bits
CS8	8 bits
CSTOPB	Send two stop bits, else one.
CREAD	Enable receiver.
PARENB	Parity enable.
PARODD	Odd parity, else even.
HUPCL	Hung up on last close.
CLOCAL	Ignore modem status lines.
TAUTOFLOW	Use hardware monitored flow control.

In addition, the input and output baud rates are also stored in the c_cflag field. The following values are supported:

Name	Description
B0	Hang up
B50	50 baud
B75	75 baud
B110	110 baud
B134	134.5 baud
B150	150 baud
B300	300 baud
B600	600 baud
B1200	1200 baud
B1800	1800 baud
B2400	2400 baud
B4800	4800 baud
B9600	9600 baud
B19200	19200 baud
B38400	38400 baud

The following interfaces are provided for getting and setting the values of the input and output baud rates in the `termios` structure. The effects on the terminal device do not become effective until the `tcsetattr()` function is successfully called.

```
speed_t cfgetospeed (termios_p)
struct termios *termios_p;

int cfsetospeed (termios_p, speed)
struct termios *termios_p;
speed_t speed;

speed_t cfgetispeed (termios_p)
struct termios *termios_p;

int cfsetispeed (termios_p, speed)
struct termios *termios_p;
speed_t speed;
```

The type speed_t is defined in <`termio.h`>.

The *termios_p* argument is a pointer to a termios structure that allows the c_cflag field to be manipulated. The cfgetospeed() returns the output baud rate stored in cflag.

Additionally, the cfsetospeed() sets the output baud rate stored in the cflag to speed. The zero baud rate, B0, is used to terminate the connection. If B0 is specified, the modem control line can no longer be asserted. Normally, this disconnects the line. Both cfsetispeed() and cfsetospeed() return a value of zero if successful and -1 to indicate error.

The cfgetispeed() returns the input baud rate stored in cflag.

The cfsetispeed() sets the input baud rate stored in cflag to speed. If the input baud rate is set to zero, the input baud rate is specified by the value of the output baud rate. For any particular hardware, unsupported baud rates are ignored. This refers to changes and baud rates not supported by the hardware, and to changes setting the input and output baud rates to different values if the hardware does not support this.

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 normally used.

If CREAD is set, the receiver is enabled. Otherwise, no characters are received.

If PARENB is set, parity generation and detection is enabled and a parity bit is added to each character. If parity is enabled, PARODD specifies odd parity, if set. Otherwise, even parity is used.

If TAUTOFLOW is set, hardware monitored flow control is performed, if the hardware supports this functionality. In this mode, the hardware terminal multiplexer suspends output upon receipt of a STOP character (ASCII DC3). The hardware resumes output after a START character (ASCII DC1) has been received. The advantage of this mode is that it provides quick response to flow control characters which would be useful in preventing overflow of the terminal device's input buffer. TAUTOFLOW is a local extension to the termios specification.

If HUPCL is set, the modem control lines for the port are lowered when the last process with the port open closes the port or the process terminates. The modem connection is broken. If HUPCL is not set, the control lines are not altered.

If CLOCAL is set, a connection does not depend on the state of the modem status lines. If CLOCAL is clear, the modem status lines are monitored. CLOCAL is typically used by direct connect terminal lines.

Under normal circumstances, a call to the open(2) function waits for the modem connection to complete. However, if the O_NONBLOCK flag is set, or, if CLOCAL has been set, the `open`(2) function returns immediately without waiting for the connection. For further information, see `open`(2). For those files on which the connection has not been established, or on which a modem disconnect has occurred, and for which CLOCAL is not set, both `read`(2) and `write`(2) return a zero character count. For `read`(2), this is equivalent to an end-of-file condition.

The initial hardware control value after open(2) is CS8, CREAD, HUPCL, B300.

Local Modes

The c_lflag field of the argument structure is used to control various functions.

Mask Name	Description
ISIG	Enable signals
ICANON	Canonical input (erase and kill processing).
NOFLSH	Disable flush after interrupt, quit, or suspend.
TOSTOP	Send SIGTTOU for background output.
ECHO	Enable echo.
ECHOE	Echo ERASE as an error-correcting backspace.
ECHOK	Echo KILL.
ECHONL	Echo 0
IEXTEN	Enable extended (implementation defined) functions.
TCTLECH	Echo input control chars as ^char, delete as ^?.
TCRTKIL	BS-space-BS erase entire line on kill.
TPRTERA	Hard-copy terminal erase mode using

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 enables the erase, word erase, reprint, and kill edit functions, and the assembly of input characters into lines delimited by NL, EOF, and EOL, as described in Canonical Mode Input Processing.

If ICANON is not set, `read`(2) requests are satisfied directly from the input queue. A `read`(2) is not satisfied until at least MIN characters have been received or the timeout value TIME expired between characters. The time value represents tenths of seconds. See the Noncanonical Mode Input Processing section for more details.

If NOFLSH is set, the normal flush of the input and output queues associated with the INTR, QUIT, and SUSP characters is not done.

If TOSTOP is set, the signal SIGTTOU is sent to the process group of a process that tries to write to its controlling terminal, if it is not in the foreground process group for that terminal. This signal, by default, stops the members of the process group. Otherwise, the output generated by that process is output to the current output stream. Processes that are holding or ignoring SIGTTOU signals are accepted and allowed to produce output and the SIGTTOU signal is not sent.

If ECHO is set, input characters are echoed back to the terminal. If ECHO is not set, input characters are not echoed.

The echo functions (ECHOE, ECHOK, ECHONL, TCTLECH, TCRTKIL, and TPRTERA) are performed if ICANON is set.

If ECHOE and ICANON are set, the ERASE character causes the terminal to erase the last character in the current line from the display, if possible.

If ECHOK and ICANON are set, the KILL character, either the terminal erases the line from the display or echoes the \n character after the KILL character.

If ECHONL and ICANON are set, the \n character is echoed even if ECHO is not set.

If IEXTEN is set, implementation defined functions are recognized from the input data. In this manner the DSUSP, RPRNT, FLUSH, WERASE, LNEXT, and QUOTE special characters in the c_cc array are only recognized if the IEXTEN flag is set.

If TCTLECH is set, all control characters are echoed as ^X, where X is the character obtained by adding the octal value of the character A (100) to the octal code for the control character. In this context, a control character is defined to be a character whose octal value is less than 37. The following control characters are excluded from TCTLECH operations: ASCII NL, ASCII TAB, as well as control characters that are defined in the c_cc array but are not returned to user programs (such as START and STOP). TCTLECH is a local extension to the local modes.

If TCRTKIL is set, the response to a kill character is to erase the present input line through a sequence of backspace-space-backspace. TCRTKIL is a local extension to the local modes.

If TCRTERA is set, characters that are logically erased are printed out backwards preceded by a backslash (\) and followed by a slash (/). This mode is useful when a hard-copy terminal is in use. TCRTERA is a local extension to the local modes.

The initial local control value after open (2) is 0 (all bits clear).

Special Control Characters

The special control characters values are defined by the array c_cc. The subscript name and description for each element in both canonical and noncanonical modes are as follows.

Subscript	Description
VINTR (INTR)	Interrupt character
VQUIT (QUIT)	Quit character
VERASE (ERASE)	Erase character
VKILL (KILL)	Kill character
VEOF (EOF)	End-of-file character
VEOL (EOL)	End-of-line character
VMIN (MIN)	Value for noncanonical reads
VTIME (TIME)	Value for noncanonical reads
VSTART (START)	Start character
VSTOP (STOP)	Stop character
VSUSP (SUSP)	Suspend character
VDSUSP (DSUSP)	Delayed suspend character
VRPRNT (RPRNT)	Reprint character
VFLUSH (FLUSH)	Flush character
VWERASE (WERASE)	Word erase character
VLNEXT (LNEXT)	Literal next character
VQUOTE (QUOTE)	Erase and kill quoting character

The following subscripts are local extensions to the c_cc array: VDSUSP, VRPRNT, VFLUSH, VWERASE, VLNEXT, and VQUOTE. The constant NCCS defines the total number of elements in the c_cc array.

Setting the value of a special character to POSIX_V_DISABLE causes that function to be disabled; that is, no input data will be recognized as the disabled special character. If ICANON is not set, the value of POSIX_V_DISABLE has no special meaning for the VMIN and VTIME entries of the c_cc array.

Line Discipline

The c_line field of the termios data structure is used to specify the line discipline. Support is provided for the basic termios line discipline only. For this reason, the value of this field should be set to the value TERMIODISC (the default value) by convention. The value of c_line is reset to TERMIODISC by the system, if attempts are made to set c_line to other values. This field is a local extension.

Control Functions

The functions that are used to control the general terminal function are described in this section. Unless otherwise noted for a specific command, these functions are restricted from use by background processes. Attempts to perform these operations cause the process group to be sent a SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process is allowed to perform the operation and the SIGTTOU signal is not sent.

In all the functions, *fildes* is an open file descriptor. However, the functions affect the underlying terminal file, not just the open instance associated with the file descriptor.

Get and Set State

Functions: `tcgetattr()`, `tcsetattr()`

Syntax

```
#include <termios.h>

int tcgetattr (fildes, termios_p)
int fildes;
struct termios *termios_p;

int tcsetattr (fildes, optional_actions, termios_p)
int fildes;
int optional_actions;
struct termios *termios_p;
```

Description

The `tcgetattr()` function retrieves the parameters associated with the object referred to by *fildes* and store them in the `termios` structure referenced by *termios_p* . This function is allowed from a background process; however, the information can be subsequently changed by a foreground process.

The `tcsetattr()` function sets the parameters associated with the terminal from the `termios` structure referenced by *termios_p* as follows:

- If *optional_actions* is TCSANOW, the change occurs immediately.
- If *optional_actions* is TCSADRAIN, the change occurs after all output written to *fildes* has been transmitted. This function should be used when changing parameters that affect output.
- If *optional_actions* is TCSADFLUSH, the change occurs after all output written to the object referred to by *fildes* has been transmitted, and all input that has been received but not read is discarded before the change is made.

Returns

Upon successful completion, a value of zero is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

Errors

If any of the following conditions occur, the `tcgetattr()` function returns –1 and sets *errno* to the corresponding value:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[EINVAL]	The device does not support the `tcgetattr()` function.
[ENOTTY]	The file associated with *fildes* is not a terminal.

If any of the following conditions occur, the `tcsetattr()` function returns –1 and sets *errno* to the corresponding value:

[EBADF]	The *fildes* argument is not a valid file descriptor.

[EINVAL] The device does not support the `tcsetatr()` function, or the *optional_actions* argument is not a proper value.

[ENOTTY] The file associated with *fildes* is not a terminal.

Line Control Functions

Functions: `tcsendbreak()`, `tcdrain()`, `tcflush()`, `tcflow()`

Syntax

```
#include <termios.h>

int tcsendbreak (fildes, duration)
int fildes;
int duration;

int tcdrain (fildes)
int fildes;

int tcflush (fildes, queue_selector)
int fildes;
int queue_selector;

int tcflow (fildes, action)
int fildes;
int action;
```

Description

The `tcsendbreak()` function sends a ''break'' that is a continuous stream of zero-valued bits for a specific duration. If duration is zero, it sends zero-valued bits for 0.25 seconds. If duration is greater than zero, it sends zero-valued bits for duration*0.1 seconds. If the object referred to by *fildes* no break sequence is generated.

The `tcdrain()` function waits until all output written to the object referred to by *fildes* has been transmitted.

The `tcflush()` function discards data written to the object referred to by *fildes* but not transmitted, or data received but not read, depending on the value of *queue_selector:*

- If *queue_selector* is TCIFLUSH, it flushes data received but not read.
- If *queue_selector* is TCOFLUSH, it flushes data written but not transmitted.
- If *queue_selector* is TCIOFLUSH, it flushes both data received but not read, and data written but not transmitted.

The `tcflow()` function suspends transmission or reception of data on the object referred to by *fildes* , depending on the value of *action:*

- If *action* is TCOOFF, it suspends output.
- If *action* is TCOON, it restarts suspended output.
- If *action* is TCIOFF, the system transmits a STOP character, which is intended to cause the terminal device to stop transmitting data to the system.
- If *action* is TCION, the system transmits a START character, which is intended to cause the terminal device to resume transmitting data to the system.
- The default on open of a terminal file is that neither its input nor its output is suspended.

Returns

Upon successful completion, a value of zero is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

Errors

If any of the following conditions occur, the `tcsendbreak()` function returns –1 and sets *errno* to the corresponding value:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[EINVAL]	The device does not support the `tcsendbreak()` function.
[ENOTTY]	The file associated with *fildes* is not a terminal.

If any of the following conditions occur, the `tcdrain()` function returns –1 and sets *errno* to the corresponding value:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[EINVAL]	The device does not support the `tcdrain()` function.
[ENOTTY]	The file associated with *fildes* is not a terminal.
[EINTR]	A signal interrupted the `tcdrain()` function.

If any of the following conditions occur, the `tcflush()` function returns –1 and sets *errno* to the corresponding value:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[EINVAL]	The device does not support the `tcflush()` function, or the *queue_selector* argument is not a proper value.
[ENOTTY]	The file associated with *fildes* is not a terminal.

If any of the following conditions occur, the `tcflow()` function returns –1 and sets *errno* to the corresponding value:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[EINVAL]	The device does not support the `tcflow()` function, or the *action* argument is not a proper value.
[ENOTTY]	The file associated with *fildes* is not a terminal.

Get Foreground Process Group Id

Function: `tcgetpgrp()`

Synopsis

```
#include <sys/types.h>
#include <termios.h>

pid_t tcgetpgrp (fildes)
int fildes;
```

Description

The `tcgetpgrp()` function returns the value of the process group ID of the foreground process group associated with the terminal.

The `tcgetpgrp()` function is allowed from a background process; however, the information can be subsequently changed by a foreground process.

Returns

Upon successful completion, `tcgetpgrp()` retuns the process group ID of the foreground process group associated with the terminal. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

Errors

If any of the following conditions occur, the `tcgetpgrp()` function returns –1 and sets *errno* to the corresponding value:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[EINVAL]	This function is not allowed for the device associated with the *fildes* argument.
[ENOTTY]	The calling process does not have a controlling terminal or the file is not the controlling terminal.

Set Foreground Process Group ID

Function: `tcsetpgrp()`

Synopsis

```
#include <sys/types.h>
#include <termios.h>

int tcsetpgrp (fildes, pgrp_id)
int fildes;
pid_t pgrp_id;
```

Description

If the process has a controlling terminal, the `tcsetpgrp()` function sets the foreground process group ID associated with the terminal to *pgrp_id*. The file associated with *fildes* must be the controlling terminal of the calling process, and the controlling terminal must be currently associated with the session of the calling process. The value of *pgrp_id* must match a process group ID of a process in the same session as the calling process.

Returns

Upon successful completion, `tcsetpgrp()` returns a value of zero. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

Errors

If any of the following conditions occur, the `tcsetpgrp()` function returns –1 and sets *errno* to the corresponding value:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[EINVAL]	This function is not allowed for the device associated with the *fildes* argument or the value of *pgrp_id* argument is less than or equal to zero, or exceeds {PID_MAX}.
[ENOTTY]	The calling process does not have a controlling terminal or the file is not the controlling terminal, or the controlling terminal is no longer associated with the session of the calling process.

[EPERM] The value of the *pgrp_id* argument does not match the process group ID of a process in the same session as the calling process.

See Also

csh(1), stty(1), tset(1), ioctl(2), sigvec(2), setsid(2), termio(4), getty(8)

Name

tms – TMSCP magnetic tape interface

Syntax

For UNIBUS, Q-bus:
```
    controller klesiu0 at uba0
    controller uq0 at klesiu0 csr 0174500 vector uqintr
    tape tms0 at uq0 drive 0
```

For MSI Bus:
```
    adapter msi0 at nexus?
    controller dssc0 at msi0 msinode 0
    tape tms0 at dssc0 drive 3
```

For VAXBI:
```
    controller klesib0 at vaxbi0 node 0
    controller uq0 at klesib0 vector uqintr
    tape tms0 at uq0 drive 0

    controller aie0 at vaxbi? node?
    controller bvpssp0 at aie0 vector bvpsspintr
    tape tms0 at bvpssp0 drive 0
```

For MSI Bus:
```
    adapter msi0 at nexus?
    controller dssc0 at msi0 msinode 0
    tape tms0 at dssc0 drive 0
```

For VAX CI/HSC:
```
    adapter ci0 at nexus?
    adapter ci0 at vaxbi? node?
    controller hsc0 at ci0 cinode 6
    tape tms0 at hsc0 drive 3
```

Description

Prior to Version 2.0, this device was referenced by tmscp(4).

The TMSCP driver provides a standard tape drive interface, as described in `mtio`(4). This is a driver for any controller that adheres to the Tape Mass Storage Control Protocol (TMSCP). The TMSCP controllers communicate with the host through a packet-oriented protocol termed the Tape Mass Storage Control Protocol. This driver also supports n-buffered reads and writes to the raw tape interface (used with streaming tape drives). See `nbuf`(4) for further details.

Tape Support

TK50, TK70, TF70, TU81, TU81e, TA78, TA79, TA81, RV20, TA90, TA90E

Diagnostics

All diagnostic messages are sent to the error logger subsystem.

Files

```
/dev/rmt???
/dev/nrmt???
```

See Also

mtio(4), nbuf(4), MAKEDEV(8), uerf(8), tapex(8)
Guide to the Error Logger System

Name

trace – system call tracing interface

Syntax

options SYS_TRACE pseudo-device sys_trace

Description

This is a special character device that provides an interface for the system call tracing facility.

Files

`/dev/trace`

See Also

MAKEDEV(8)

ts(4)

Name

ts – TS11/TS05/TU80 magnetic tape interface

Syntax

controller zs0 at uba? csr 0172520 vector tsintr
tape ts0 at zs0 drive 0

Description

The TS11 combination provides a standard tape drive interface as described in `mtio`(4). The TS11 operates only at 1600 bpi, and only one transport is possible per controller.

Diagnostics

ts%d: no write ring
An attempt was made to write on the tape drive when no write ring was present. This message is written on the terminal of the user who tried to access the tape.

ts%d: not online
An attempt was made to access the tape while it was off line. This message is written on the terminal of the user who tried to access the tape.

ts%d: hard error bn%d
A hard error occurred on the tape at block *bn*. Additional register information may be gathered from the system error log file,
`/usr/adm/syserr/syserr.<hostname>`.

Files

```
/dev/rmt???
/dev/nrmt???
```

See Also

mtio(4), nbuf(4), MAKEDEV(8), uerf(8)

Name

tty – general terminal interface

Description

Terminal Subsystem

The terminal subsystem is the part of the operating system that allows users to read and write characters over asynchronous terminal lines. An important aspect of this subsystem is to provide a means for the user to set and receive terminal attributes. Terminal attributes involve such things as line speed (baud rate), character length, parity, flow control, modem control, as well as numerous character-processing capabilities.

The ULTRIX terminal interface allows the user to specify terminal attributes in different ways. Several different terminal interfaces exist to provide standard compliant terminal control.

Terminal Interface Definitions

The `tty`(4) special file describes the standard Berkeley terminal interface. This interface is backward compatible with earlier versions of the ULTRIX operating system.

The `termio`(4) special file describes the terminal interface as defined by the System V Interface Definition.

The `termios`(4) special file describes the termios termio interface as defined by the IEEE P1003 POSIX specification.

Functionally, the three terminal interfaces are quite similar. Major differences lie in how terminal attributes are specified. This includes the use of different data structures to represent terminal attributes, as well as the means of setting and receiving attributes.

It is possible to use combinations of the three terminal interface definitions. Under these circumstances, the attributes of one interface are mapped into the corresponding attributes of the other interfaces. Combining aspects of different interfaces is discouraged, because it prevents the development of portable programs. For example, a program intended for use with the System V termio terminal interface fails to be a standard compliant program if it sets terminal attributes that are specific to either of the other two terminal interfaces.

Combinations of the different terminal interfaces should be used with extreme caution to avoid unwanted side-effects. For example, a program may have initially set up its terminal environment using the System V termio interface, `termio`(4). Suppose that the initial line settings are seven bits even parity with input and output processing performed. If this same program were to set the line to RAW mode as specified in `tty`(4), the line would be set to eight bits no parity, with no input or output processing. These settings would be reflected in the parameters as specified in `termio`(4). This simple example is meant to illustrate the subtle side effects that can result from the use of combinations of terminal interfaces.

Using combinations of terminal interfaces can also cause problems if attributes that are not common to all interfaces are used. For example, the Berkeley terminal interface allows the user to set the value of the start and stop characters. The System V termio interface defines that the start and stop characters shall be Control-Q and Control-S. As a result, if a terminal changes the start and stop characters using the Berkeley terminal interface, those characters are reset if the terminal parameters are set using the System V termio interface.

Terminal Interface Usage

The three interfaces have been developed to provide standard compliant terminal behavior. The interface type should be specified at the time of program compilation. As described in `cc`(1), to compile a System V-compliant program, the -Y option (or setting PROG_ENV equal to SYSTEM_FIVE) should be used. Similarly, the -YPOSIX option should be used to compile a POSIX-compliant program. Without the -Y or -YPOSIX compile option, the program intends to use the Berkeley terminal interface. Refer to `intro`(2) for specific details on compatibility modes.

Berkeley Terminal Interface

Line Disciplines.

The system provides different *line disciplines* for controlling communications lines. In this version of the system, there are several disciplines available:

old The old (Version 7) terminal driver. This is sometimes used when using the standard shell `sh`(1) and for compatibility with other standard Version 7 UNIX systems.

new The standard terminal driver, with features for job control. This must be used when using `csh`(1).

net A line discipline used for networking and loading data into the system over communications lines. It allows high speed input at low overhead and is described in `bk`(4).

termio This line discipline is intended for use by System V programs that use the termio interface, as described in `termio`(4). The termio line discipline is also used by programs that require a POSIX IEEE P1003 termios interface as described in `termios`(4).

Line discipline switching is accomplished with the TIOCSETD `ioctl`:

int ldisc = LDISC; ioctl(filedes, TIOCSETD, &ldisc);

LDISC is OTTYDISC for the standard tty driver, NTTYDISC for the new driver, NETLDISC for the networking discipline, and TERMIODISC for System V termio and POSIX termios. The standard tty driver is discipline 0 by convention. Other disciplines may exist for special purposes. The current line discipline can be obtained with the TIOCGETD ioctl. Pending input is discarded when the line discipline is changed.

All of the low-speed asynchronous communications ports can use any of the available line disciplines, no matter what hardware is involved. The remainder of this section discusses the old and new disciplines.

The Control Terminal

When a terminal file is opened, it causes the process to wait until a connection is established. In practice, user programs seldom open these files; they are opened by `getty`(8) or `rlogind`(8c) and become a user's standard input and output file.

If a process that has no control terminal opens a terminal file, then that terminal file becomes the control terminal for that process. The control terminal is thereafter inherited by a child process during a `fork`(2), even if the control terminal is closed.

The file `/dev/tty` is, in each process, a synonym for a *control terminal* associated with that process. It is useful for programs that want 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 a file name for output, when typed output is desired and it is tiresome to find out which terminal is currently in use.

A process can remove the association it has with its controlling terminal by opening the file `/dev/tty` and issuing:

ioctl(fildes, TIOCNOTTY, 0)

This is often desirable in server processes.

Process Groups

Command processors such as `csh`(1) can arbitrate the terminal between different *jobs* by placing related jobs in a single process group and associating this process group with the terminal. A terminal's associated process group may be set using the TIOCSPGRP `ioctl`(2):

ioctl(fildes, TIOCSPGRP, &pgrp)

The process group can be examined using using TIOCGPGRP, rather than TIOCSPGRP, returning the current process group in *pgrp*. The new terminal driver aids in this arbitration by restricting access to the terminal by processes that are not in the current process group; see **Job Access Control.**

Modes

The terminal drivers have three major modes, characterized by the amount of processing on the input and output characters:

cooked The normal mode. In this mode, lines of input are collected and input editing is done. The edited line is made available when it is completed by a newline or when the *t_brkc* character (normally undefined) or *t_eofc* character (normally an EOT, CTRL/D) is entered. A carriage return is usually made synonymous with newline in this mode and replaced with a newline whenever it is typed. All driver functions (input editing, interrupt generation, output processing such as delay generation and tab expansion, and so forth) are available in this mode.

CBREAK This mode eliminates the character, word, and line editing input facilities, making the input character available to the user program as it is typed. Flow control, literal-next, and interrupt processing are still done in this mode. Output processing is done.

RAW This mode eliminates all input processing and makes all input characters available as they are typed; no output processing is done either.

The style of input processing can also be different when the terminal is put in nonblocking I/O mode. For further information, see the FNDELAY flag described in `fcntl`(2). In this case, a `read`(2) from the control terminal never blocks. Rather, it returns an error indication (EWOULDBLOCK), if there is no input available.

A process may also request a SIGIO signal be sent it whenever input is present and also whenever output queues fall below the low-water mark. To enable this mode, the FASYNC flag should be set using `fcntl`(2).

Input Editing

An ULTRIX terminal 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 choked, 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 256 characters. In RAW mode, the terminal driver throws away all input and output without notice when the limit is reached. In CBREAK mode or cooked mode, it refuses to accept any further input and, if in the new line discipline, rings the terminal bell.

Input characters are normally accepted in either even or odd parity, with the parity bit being stripped off before the character is given to the program. By clearing either the EVEN or ODD bit in the flags word, it is possible to have input characters with that parity discarded (see the **Summary**).

In all of the line disciplines, it is possible to simulate terminal input using the TIOCSTI ioctl, 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 control terminal, except for the superuser. (This call is not in standard Version 7 UNIX.)

Input characters are normally echoed by putting them in an output queue as they arrive. This may be disabled by clearing the ECHO bit in the flags word using the `stty`(3) call or the TIOCSETN or TIOCSETP ioctl (see the **Summary**).

In cooked mode, terminal input is processed in units of lines. A program attempting to read is normally suspended until an entire line has been received (but, see the description of SIGTTIN in **Job Access Control** and FIONREAD in **Summary of modes**.) No matter how many characters are requested in the read call, at most one line is returned. It is not, however, necessary to read a whole line at once; any number of characters can be requested in a read, even one, without losing information.

During input, line editing is normally done, with the erase character *sg_erase* (by default, the number sign (#)) logically erasing the last character typed and the *sg_kill* character (by default, the at sign (@)) logically erasing the entire current input line. These are often reset on CRTs, with CTRL/H replacing the number sign (#), and CTRL/U replacing the at sign (@). These characters never erase beyond the beginning of the current input line or an EOF. These characters may be entered literally, by preceding them with a backslash (\). In the old Teletype driver, both the backslash (\) and the character entered literally appear on the screen; in the new driver, the (\) normally disappears.

The drivers normally treat either a carriage return or a newline character as terminating an input line, replacing the return with a newline and echoing a return and a line feed. If the CRMOD bit is cleared in the local mode word, then the processing for carriage return is disabled, and it is simply echoed as a return and does not terminate cooked mode input.

In the new driver, there is a literal-next character (normally CTRL/V), which can be typed in both cooked and CBREAK mode preceding any character to prevent its special meaning to the terminal handler. This is to be preferred to the use of the backslash (\) escaping erase and kill characters, but the backslash (\) is retained with its old function in the new driver for historical reasons.

The new terminal driver also provides two other editing characters in normal mode. The word-erase character, normally CTRL/W, erases the preceding word, but not any spaces before it. For the purposes of CTRL/W, a word is defined as a sequence of nonblank characters, with tabs counted as blanks. Finally, the reprint character, normally CTRL/R, retypes the pending input beginning on a new line. Retyping occurs automatically in cooked mode, if characters that would normally be erased from the screen are fouled by program output.

Input Echoing and Redisplay

In the old terminal driver, the erase character is simply echoed. When a kill character is typed, it is echoed, followed by a newline (even if the character is not killing the line, because it was preceded by a backslash (\).)

The new terminal driver has several modes for handling the echoing of terminal input, controlled by bits in a local mode word.

Hardcopy terminals. When a hardcopy terminal is in use, the LPRTERA bit is normally set in the local mode word. Characters that are logically erased are then printed out backwards, preceded by a backslash (\) and followed by a slash (/) in this mode.

CRT terminals When a CRT terminal is in use, the LCRTBS bit is normally set in the local mode word. The terminal driver echoes the proper number of erase characters when input is erased. In the normal case, where the erase character is a CTRL/H, this causes the cursor of the terminal to back up to where it was before the logically erased character was typed. If the input has become fouled due to interspersed asynchronous output, the input is automatically retyped.

Erasing characters from a CRT When a CRT terminal is in use, the LCRTERA bit can be set to cause input to be erased from the screen with a ''backspace-space-backspace'' sequence when character- or word-deleting sequences are used. LCRTERA must be used with LCRTBS for this functionality. A LCRTKIL bit can be set as well, causing the input to be erased in this manner on line kill sequences as well.

Echoing of control characters If the LCTLECH bit is set in the local state word, then nonprinting (control) characters are normally echoed as ^*x* (where *x* is the character used in combination with the CTRL key), rather than being echoed unmodified; delete is echoed as `^?`.

The normal modes for using the new terminal driver on CRT terminals are speed-dependent. At speeds less than 1200 baud, the LCRTERA and LCRTKILL processing is slow, so `stty`(1) normally just sets LCRTBS and LCTLECH; at speeds of 1200 baud or greater, all of these bits are normally set. The `stty` command summarizes these option settings and the use of the new terminal driver as ''newcrt''.

Output Processing

When one or more characters are written, they are actually transmitted to the terminal as soon as previously written characters have finished typing. (As noted above, input characters are normally echoed by putting them in the output queue as they arrive.) When a process produces characters more rapidly than they can be typed, it is suspended when its output queue exceeds some limit. When the queue has drained down to some threshold, the program is resumed. Even parity is normally generated on output. The EOT character is not transmitted in cooked mode, to prevent terminals that respond to it from hanging up; programs using RAW or CBREAK mode should be careful.

The terminal drivers provide necessary processing for cooked and CBREAK mode output including delay generation for certain special characters and parity generation. Delays are available after backspaces (CTRL/H), form feeds (CTRL/L), carriage returns (CTRL/M), tabs (CTRL/I), and newlines (CTRL/J). The driver also optionally expands tabs into spaces, where the tab stops are assumed to be set every eight columns. These functions are controlled by bits in the tty flags word. (See **Summary**.)

The terminal drivers provide for mapping between uppercase and lowercase on terminals lacking lowercase, and for other special processing on deficient terminals.

Finally, in the new terminal driver, there is an output flush character, normally CTRL/O, which sets the LFLUSHO bit in the local mode word, causing subsequent output to be flushed until it is cleared by a program or more input is typed. This character has effect in both cooked and CBREAK modes and causes pending input to be retyped if there is any pending input. An ioctl to flush the characters in the input and output queues, TIOCFLUSH, is also available.

Uppercase Terminals and Hazeltines

If the LCASE bit is set in the tty flags, then all uppercase letters are mapped into the corresponding lowercase letter. The uppercase letter may be generated by preceding it by a backslash (\). If the new terminal driver is being used, then uppercase letters are preceded by a a backslash (\) when output. In addition, the following escape sequences can be generated on output and accepted on input:

for	`	|	~	{	}
use	\´	\!	\^	\(	\)

To deal with Hazeltine terminals, which do not recognize the tilde (~) as an ASCII character, the LTILDE bit may be set in the local mode word when using the new terminal driver; in this case, the tilde (~) will be replaced with the grave accent (`) on output.

Flow Control

There are two characters (the stop character, normally CTRL/S, and the start character, normally CTRL/Q), that cause output to be suspended and resumed respectively. Extra stop characters typed when output is already stopped have no effect, unless the start and stop characters are made the same, in which case output resumes.

A bit in the flags word may be set to put the terminal into TANDEM mode. In this mode, the system produces a stop character (default CTRL/S) when the input queue is in danger of overflowing, and a start character (default CTRL/Q) when the input has drained sufficiently. This mode is useful when the terminal is actually another machine that obeys the conventions.

A bit in the local mode word may be set to put the terminal into AUTOFLOW mode. In this mode, flow control characters are responded to at the hardware level. Upon receipt of a stop character, the hardware suspends output. This allows for quick response to the stop character, which prevents buffer overflow (in printers for example). AUTOFLOW functionality is only provided if the start character is CTRL/Q and the stop character is CTRL/S. The AUTOFLOW bit is cleared if the start or stop characters are not standard values, or if the RAW bit is not set.

Line Control and Breaks

There are several `ioctl` calls available to control the state of the terminal line. The TIOCSBRK ioctl sets the break bit in the hardware interface, causing a break condition to exist. This can be cleared by TIOCCBRK, usually after a delay with `sleep`(3). Break conditions in the input are reflected as a null character in RAW mode or as the interrupt character in cooked or CBREAK mode. The TIOCCDTR ioctl clears the data terminal ready condition. It can be set again by TIOCSDTR.

When the carrier signal from the dataset drops (usually because the user has hung up his terminal), a SIGHUP hangup signal is sent to the processes in the distinguished process group of the terminal. This usually causes them to terminate (the SIGHUP can be suppressed by setting the LNOHANG bit in the local state word of the driver.) Access to the terminal by other processes is then normally revoked, so any further reads fail, and programs that read a terminal and test for end-of-file on their input terminate appropriately.

When using an ACU, it is possible to ask that the phone line be hung up on the last close with the TIOCHPCL ioctl. This is normally done on the outgoing line.

Interrupt Characters

There are several characters that generate interrupts in cooked and CBREAK mode. All are sent the processes in the control group of the terminal, as if a TIOCGPGRP ioctl were done to get the process group and then a `killpg`(2) system call were done, except that these characters also flush pending input and output when typed at a terminal (for example, TIOCFLUSH). The characters shown here are the defaults. The field names in the structures are also shown. The characters may be changed, although this is not often done.

^?	**t_intrc** (Delete) generates a SIGINT signal. This is the normal way to stop a process that is no longer interesting or to regain control in an interactive program.
^\	**t_quitc** (FS) generates a SIGQUIT signal. This is used to cause a program to terminate and produce a core image, if possible, in the file **core** in the current directory.
^Z	**t_suspc** (EM) generates a SIGTSTP signal that is used to suspend the current process group.
^Y	**t_dsuspc** (SUB) generates a SIGTSTP signal as CTRL/Z does, but the signal is sent when a program attempts to read the CTRL/Y, rather than when it is typed.

Job Access Control

When using the new terminal driver, if a process that is not in the distinguished process group of its control terminal attempts to read from that terminal, its process group is sent a SIGTTIN signal. This signal normally causes the members of that process group to stop. If, however, the process is ignoring SIGTTIN, has SIGTTIN blocked, is an *orphan process*, or is in the middle of process creation using `vfork`(2), it is returned an end-of-file instead. (An *orphan process* is a process whose parent has exited and that has been inherited by the `init`(8) process.) Under older UNIX systems these processes would typically have had their input files reset to `/dev/null`, so this is a compatible change.

When using the new terminal driver with the LTOSTOP bit set in the local modes, a process is prohibited from writing on its control terminal, if it is not in the distinguished process group for that terminal. Processes that are holding or ignoring SIGTTOU signals, that are orphans, or that are in the middle of a `vfork2` are excepted and allowed to produce output.

Modem Control

Ioctls have been added to provide more flexible modem control on tty lines. The new commands are summarized below.

TIOCMODEM Indicate to the system that this tty line has a modem attached to it and should not ignore modem signals. The argument to this ioctl is the address of a word that contains either zero or a nonzero value. Zero indicates that the effect of the ioctl is temporary, and the line is reset to its condition prior to the ioctl, when the tty line is closed. Nonzero indicates that the effect of the ioctl should be permanent. Root privilege is required to effect a permanent change.

TIOCNMODEM
Indicate to the system that modem transmissions should be ignored on this line. This is useful for connections that do not implement the full RS-232 standard (most direct connections to terminals). The argument to this ioctl is the address of a word that contains either zero or a nonzero value. Zero indicates that the effect of ioctl is temporary, and the line is reset to its condition prior to the ioctl, when the tty line is closed. Nonzero indicates that the effect of the ioctl should be permanent. Root privilege is required to effect a permanent change.

TIOCNCAR Ignore soft carrier when doing reads or writes. If carrier is not present on a modem line, then reads or writes normally fail. This ioctl allows reads and writes to succeed, regardless of the state of this line. This is useful for dealing with automatic call units that send status messages before carrier is present on the line. The alternative would be to use the TIOCNMODEM ioctl and ignore all modem signals and force soft carrier to be present. The latter alternative is not desirable, if full modem control is required.

TIOCCAR The opposite effect of TIOCNCAR. If carrier is not present on modem lines, then reads and writes fail.

TIOCWONLINE This ioctl blocks the process until carrier is detected.

The following example demonstrates how one might deal with a modem:

```
/* open the line and don't wait for carrier */
fd = open(dcname, O_RDWR|O_NDELAY);
/* we are attached to a modem so don't ignore modem signals */
ioctl(fd, TIOCMODEM, &temp);
ioctl(fd, TIOCNCAR);       /* ignore soft carr while dialing number */
/*
 * dial phone number and negotiate with auto call unit.
 */
ioctl(fd, TIOCCAR);        /* don't ignore carrier anymore */
alarm(40);
ioctl(fd, TIOCWONLINE); /* wait for carrier */
alarm(0);
```

Shared tty Lines

The following ioctls are used by `getty`(8), `tip`(1), and `uucp`(1) to implement shared terminal lines: TIOCSINUSE/FIOSINUSE, TIOCCINUSE/FIOCINUSE. Shared terminal lines can be used for both incoming and outgoing connections. For further information, see the *Guide to System Environment Setup*. These ioctls can be used by any user process on any file type, but they do not work on a socket.

TIOCSINUSE TIOCSINUSE is defined to FIOSINUSE. This command checks to see if the file is marked ''in use''. If the file is not ''in use'', it is marked ''in use'' by the current process and the ioctl succeeds. If the file is already ''in use'' by some other process, the ioctl fails and errno is set to EALREADY. For further information, see `open`(2).

TIOCCINUSE TIOCCINUSE is defined to FIOCINUSE. This command clears the ''in use'' flag on a file, if the current process was the one that set the ''in use'' flag. Any process that is blocked and waiting for the ''in use'' flag to clear will be resumed. For further information, see `open`(2).

Summary of Modes

Unfortunately, due to the evolution of the terminal driver, there are four different structures that contain various portions of the driver data. The first of these (**sgttyb**) contains that part of the information largely common between Version 6 and Version 7 UNIX systems. The second contains additional control characters added in Version 7. The third is a word of local state peculiar to the new terminal driver, and the fourth is another structure of special characters added for the new driver.

Basic modes: sgtty – There are two versions of the sgttyb structure: one for BSD (default) and one for SYSTEM_FIVE. The basic *ioctl*s use the structure defined in `<sgtty.h>`:

You get this version of sgttyb if you include `<sgtty.h>`, into your .c source, and then compile with 'cc –YBSD ....' or 'setenv PROG_ENV BSD' or by default to BSD if PROG_ENV is not defined, or '–Y' is not specified.

```
struct sgttyb {
       char     sg_ispeed;
       char     sg_ospeed;
       char     sg_erase;
       char     sg_kill;
```

```
            short    sg_flags;
    };
```

You get this version of sgttyb if you include `<sgtty.h>` in your .c source, and then compile using the '–Y' or '–YSYSTEM_FIVE' option of `cc`, or set PROG_ENV environment to 'SYSTEM_FIVE'.

```
    struct sgttyb {
            char     sg_ispeed;
            char     sg_ospeed;
            char     sg_erase;
            char     sg_kill;
            int      sg_flags;
    };
```

The *sg_ispeed* and *sg_ospeed* fields describe the input and output speeds of the device according to the following table, which corresponds to the speeds offered on most Digital terminal multiplexers. If other hardware is used, impossible speed changes are ignored. Symbolic values in the table are as defined in `<sgtty.h>`.

B0	0	(hang up dataphone)
B50	1	50 baud
B75	2	75 baud
B110	3	110 baud
B134	4	134.5 baud
B150	5	150 baud
B200	6	200 baud
B300	7	300 baud
B600	8	600 baud
B1200	9	1200 baud
B1800	10	1800 baud
B2400	11	2400 baud
B4800	12	4800 baud
B9600	13	9600 baud
EXTA	14	External A (19200 baud)
EXTB	15	External B (38400 baud)

Code conversion and line control required for IBM 2741s (134.5 baud) must be implemented by the user's program. The half-duplex line discipline required for the 202 dataset (1200 baud) is not supplied; full-duplex 212 datasets work fine.

The *sg_erase* and *sg_kill* fields of the argument structure specify the erase and kill characters respectively. (Defaults are the number sign (#) and the at sign (@).)

The *sg_flags* field of the argument structure contains several bits that determine the system's treatment of the terminal:

ALLDELAY	0177400	Delay algorithm selection
BSDELAY	0100000	Select backspace delays (not implemented):
BS0	0	
BS1	0100000	
VTDELAY	0040000	Select form-feed and vertical-tab delays:
FF0	0	
FF1	0100000	

CRDELAY	0030000	Select carriage-return delays:
CR0	0	
CR1	0010000	
CR2	0020000	
CR3	0030000	

TBDELAY	0006000	Select tab delays:
TAB0	0	
TAB1	0002000	
TAB2	0004000	
XTABS	0006000	

NLDELAY	0001400	Select new-line delays:
NL0	0	
NL1	0000400	
NL2	0001000	
NL3	0001400	

EVENP	0000200	Even parity allowed on input (most terminals)
ODDP	0000100	Odd parity allowed on input
RAW	0000040	Raw mode: wake up on all characters, 8-bit interface
CRMOD	0000020	Map CR into LF; echo LF or CR as CR-LF
ECHO	0000010	Echo (full duplex)
LCASE	0000004	Map uppercase to lowercase on input
CBREAK	0000002	Return each character as soon as typed
TANDEM	0000001	Automatic flow control

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.

Backspace delays are ignored but might be used for Terminet 300s.

If a form-feed/vertical tab delay is specified, it lasts for about two seconds.

Carriage-return delay type 1 lasts about .08 seconds and is suitable for the Terminet 300. Delay type 2 lasts about .16 seconds and is suitable for the VT05 and the TI 700. Delay type 3 is suitable for the Concept-100 and pads lines to be at least nine characters at 9600 baud.

New-line delay type 1 is dependent on the current column and is tuned for Teletype Model 37s. Type 2 is useful for the VT05 and is about .10 seconds. Type 3 is is 0 and is unimplemented.

Tab delay type 1 is dependent on the amount of movement and is tuned to the Teletype Model 37. Type 3, called XTABS, is not a delay at all but causes tabs to be replaced by the appropriate number of spaces on output.

The flags for even and odd parity control parity checking on input and generation on output in cooked and CBREAK mode. Even parity is generated on output unless ODDP is set and EVENP is clear, in which case odd parity is generated. For no parity, set both ODDP and EVENP flags. Input characters with the wrong parity, as determined by EVENP and ODDP, are ignored in cooked and CBREAK mode.

RAW disables all processing save output flushing with LFLUSHO; full eight bits of input are given as soon as it is available; all eight bits are passed on output. A break condition in the input is reported as a null character. If the input queue overflows in raw mode, it is discarded; this applies to both new and old drivers.

CRMOD causes input carriage returns to be turned into newlines; input of either CR or LF causes LF-CR both to be echoed (for terminals with a newline function).

CBREAK is a sort of half-cooked mode. Programs can read each character as soon as typed, instead of waiting for a full line; all processing is done, except the input editing: character and word erase and line kill, input reprint, and the special treatment of the backslash (\) or EOT are disabled.

TANDEM mode causes the system to produce a stop character (default, CTRL/S), whenever the input queue is in danger of overflowing, and a start character (default CTRL/Q), when the input queue has drained sufficiently. It is useful for flow control when the ''terminal'' is really another computer that understands the conventions.

Basic ioctls – In addition to the TIOCSETD and TIOCGETD disciplines discussed in **Line disciplines**, a large number of other `ioctl`(2) calls apply to terminals and have the general form:

#include <sgtty.h>

ioctl(fildes, code, arg)
struct sgttyb *arg;

The applicable codes are:

TIOCGETP	Fetch the basic parameters associated with the terminal and store in the pointed-to *sgttyb* structure.
TIOCSETP	Set the parameters according to the pointed-to *sgttyb* structure. The interface delays until output is quiescent, and then throws away any unread characters, before changing the modes.
TIOCSETN	Set the parameters like TIOCSETP but do not delay or flush input. Input is not preserved, however, when changing to or from RAW.

With the following codes the *arg* is ignored.

TIOCEXCL	Set ''exclusive-use'' mode: all open calls to this line have been closed. This setting does not prevent superuser opens of the terminal line.
TIOCNXCL	Turn off ''exclusive-use'' mode.
TIOCHPCL	When the file is closed for the last time, hang up the terminal. This is useful when the line is associated with an ACU used to place outgoing calls.

Setting the pointed-to integer parameter to the following values determines how TIOCFLUSH functions.

TIOCFLUSH	FREAD flushes input queues. FWRITE flushes output queues. Zero (0) flushes both. FREAD and FWRITE are defined in `<sys/file.h>`.

In cases where arguments are required, they are described; *arg* should otherwise be given as zero (0).

TIOCSTI	The argument is the address of a character that the system pretends was typed on the terminal.
TIOCSBRK	The break bit is set in the terminal.
TIOCCBRK	The break bit is cleared.
TIOCSDTR	Data terminal ready is set.
TIOCCDTR	Data terminal ready is cleared.
TIOCSTOP	Output is stopped, as if the ''stop'' character had been typed.
TIOCSTART	Output is restarted, as if the ''start'' character had been typed.
TIOCGPGRP	*arg* is the address of a word into which is placed the process group number of the control terminal.
TIOCSPGRP	*arg* is a word (typically a process ID) that becomes the process group for the control terminal.
FIONREAD	Returns in the long integer whose address is *arg,* the number of immediately readable characters from the argument unit.

Tchars – The second structure associated with each terminal specifies characters that are special in both the old and new terminal interfaces. The following structure is defined in `<sys/ioctl.h>`, which is automatically included in `<sgtty.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 default values for these characters are CTRL/?, CTRL/\, CTRL/Q, CTRL/S, CTRL/D, and –1. A character value of –1 eliminates the effect of that character. The *t_brkc* character, by default –1, acts like a newline in that it terminates a ''line'', is echoed, and is passed to the program. The ''stop'' and start characters may be the same, to produce a toggle effect. It is probably counterproductive to make other special characters (including erase and kill) identical. The applicable ioctl calls are:

TIOCGETC	Get the special characters and put them in the specified structure.
TIOCSETC	Set the special characters to those given in the structure.

Local mode – The third structure associated with each terminal is a local mode word. The bits of the local mode word are:

LCRTBS	0x0001	Backspace on erase, rather than echoing erase
LPRTERA	0x0002	Printing terminal erase mode
LCRTERA	0x0004	Erase character echoes as backspace-space-backspace
LTILDE	0x0008	Convert ~ to ` on output (for Hazeltine terminals)
LLITOUT	0x0020	Suppress output translations
LTOSTOP	0x0040	Send SIGTTOU for background output
LFLUSHO	0x0080	Output is being flushed
LNOHANG	0x0100	Do not send hangup when carrier drops
LAUTOFLOW	0x0200	Hardware responds to flow control characters. (See **Flow control.**)
LCRTKIL	0x0400	BS-space-BS erase entire line on line kill

LPASS8	0x0800	Allow 8-bit characters in input and output
LCTLECH	0x1000	Echo input control chars as ^X, delete as ^?
LPENDIN	0x2000	Retype pending input at next read or input character
LDECCTQ	0x4000	Only CTRL/Q restarts output after CTRL/S
LNOFLSH	0x8000	Do not flush output on receipt of suspend or interrupt character

The applicable `ioctl` functions are:

TIOCLBIS *arg* is the address of a mask of bits to be set in the local mode word.

TIOCLBIC *arg* is the address of a mask of bits to be cleared in the local mode word.

TIOCLSET *arg* is the address of a mask to be placed in the local mode word.

TIOCLGET *arg* is the address of a word into which the current mask is placed.

Window Size – The fourth structure associated with terminals is the `winsize` structure that defines the size of the terminal window. The `winsize` structure is defined as follows:

```
struct winsize {
        unsigned short  ws_row, ws_col;
        unsigned short  ws_xpixel, ws_ypixel;
};
```

The *ws_row* and *ws_col* elements define the window size in terms of the number of characters per row and column respectively. The *ws_xpixel* and *ws_ypixel* define the window size in terms of pixels. The default value is to initialize each of the elements to zero.

The applicable `ioctl` functions are:

TIOCSWINSZ *arg* is the address of a `winsize` structure, which defines the new window sizes. This will send a SIGWINCH signal to notify all members of process group that the window size has changed.

TIOCGWINSZ *arg* is the address of a `winsize` structure into which is placed the current window size settings.

Local special characters – The final structure associated with each terminal is the `ltchars` structure that defines interrupt characters for the new terminal driver. 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 default values for these characters are CTRL/Z, CTRL/Y, CTRL/R, CTRL/O, CTRL/W, and CTRL/V. A value of –1 disables the character.

The applicable ioctl functions are:

TIOCSLTC *args* is the address of an `ltchars` structure, which defines the new local special characters.

TIOCGLTC *args* is the address of an `ltchars` structure, into which is placed the current set of local special characters.

Restrictions

Half-duplex terminals are not supported.

Files

```
/dev/tty
/dev/tty*
/dev/console
```

See Also

csh(1), stty(1), tset(1), ioctl(2), sigvec(2), stty(3), termio(4), termios(4), getty(8), MAKEDEV(8)

Name

tu – TM03/TE16/TU45/TU77 magnetic tape interface

Syntax

master ht0 at mba? drive ?
tape tu0 at ht0 slave 0

Description

Prior to Version 2.0, this device was referenced by `ht`(4).

The TM03/transport combination provides a standard tape drive interface, as described in `mtio`(4). All drives provide both 800 and 1600 bpi. The TE16 runs at 45 ips, the TU45 at 75 ips, and the TU77 runs at 125 ips and autoloads tapes.

Diagnostics

tu%d: no write ring
An attempt was made to write on the tape drive when no write ring was present. This message is written on the terminal of the user who tried to access the tape.

tu%d: not on line
An attempt was made to access the tape while it was off line. This message is written on the terminal of the user who tried to access the tape.

tu%d: can't switch density in mid-tape
An attempt was made to write on a tape at a different density than is already recorded on the tape. This message is written on the terminal of the user who tried to switch the density.

tu%d: hard error bn%d
A tape error occurred at block *bn*. Any error is fatal on nonraw tape. When possible, the driver will have retried the operation and failed several times before reporting the error. Additional register information can be gathered from the system error log file, `/usr/adm/syserr/syserr.`*<hostname>*.

Files

```
/dev/rmt???
/dev/nrmt???
```

See Also

mtio(4), nbuf(4), MAKEDEV(8), uerf(8)

Name

tz – SCSI magnetic tape interface

Syntax

```
VAX NCR 5380:
   adapter         uba0      at nexus?
   controller      scsi0     at uba0       csr 0x200c0080  vector szintr
   tape            tz0       at scsi0      drive 0
VAX DEC SII:
   adapter         ibus0     at nexus?
   controller      sii0      at ibus?      vector sii_intr
   tape            tz0       at sii0       drive 0
RISC DEC SII:
   adapter         ibus0     at nexus?
   controller      sii0      at ibus?      vector sii_intr
   tape            tz0       at sii0       drive 0
RISC NCR ASC:
   adapter         ibus0     at nexus?
   controller      asc0      at ibus?      vector ascintr
   tape            tz0       at asc0       drive 0
```

Description

The SCSI tape driver provides a standard tape drive interface as described in `mtio`(4). This is a driver for any Digital SCSI tape device. This driver also supports n-buffered reads and writes to the raw tape interface (used with streaming tape drives). See `nbuf`(4) for further details.

Tape Support

TZ30, TZK50, TLZ04, TSZ05

Diagnostics

All diagnostic messages are sent to the error logger subsystem.

Files

```
/dev/rmt???
/dev/nrmt???
```

See Also

mtio(4), nbuf(4), SCSI(4), MAKEDEV(8), uerf(8), tapex(8)
Guide to the Error Logger System

Name

udp – Internet User Datagram Protocol

Syntax

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

s = socket(AF_INET, SOCK_DGRAM, 0);

Description

UDP is a simple, unreliable datagram protocol that is used to support the SOCK_DGRAM abstraction for the Internet protocol family. UDP sockets are connectionless and are normally used with the `sendto` and `recvfrom` calls, though the `connect`(2) call can also be used to fix the destination for future packets (in which case the `recv`(2) or `read`(2) and `send`(2) or `write`(2) system calls may be used).

UDP address formats are identical to those used by TCP. In particular, UDP provides a port identifier in addition to the normal Internet address format. Note that the UDP port space is separate from the TCP port space (for example, a UDP port may not be ''connected'' to a TCP port). In addition broadcast packets can be sent (assuming the underlying network supports this) by using a reserved ''broadcast address''; this address is network interface dependent. The SO_BROADCAST option must be set on the socket for broadcasting to succeed.

Diagnostics

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

[EISCONN]	Try 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 already connected.
[ENOTCONN]	Try to send a datagram, but no destination address is specified, and the socket has not been connected.
[ENOBUFS]	The system runs out of memory for an internal data structure.
[EADDRINUSE]	An attempt is made to create a socket with a port that has already been allocated.
[EADDRNOTAVAIL]	An attempt is made to create a socket with a network address for which no network interface exists.

See Also

getsockopt(2), send(2), socket(2) recv(2), intro(4n), inet(4f)

Name

xna – DEBNI and DEMNA Ethernet interfaces

Syntax

device xna0 at vaxbi? node? vector xnaintr (DEBNI)
device xna0 at xmi? node? vector xnaintr (DEMNA)

Description

The `xna` driver provides access to a 10 Mbytes Ethernet network through the DEBNI and DEMNA adapters. The DEBNI is an Ethernet to BI bus. The DEMNA is an Ethernet to XMI adapter.

The host's Internet address is specified at boot time with an SIOCSIFADDR ioctl. The `xna` driver employs the Address Resolution Protocol, as described in `arp`(4p), to map dynamically between Internet and Ethernet addresses on the local network.

The `xna` driver normally tries to use a trailer encapsulation to minimize copying data on input and output. This can be disabled for an interface by setting the IFF_NOTRAILERS flag with an SIOCSIFFLAGS ioctl. Trailers are used only for packets destined for Internet hosts.

The SIOCSPHYSADDR ioctl can be used to change the physical address of the adapater and SIOCRPHYSADDR can be used to read its physical address. SIOCADDMULTI and SIOCDELMULTI can be used to add or delete multicast addresses. The `xna` driver supports a maximum of 12 multicast addresses. The argument to the latter ioctls is a pointer to an ''ifreq'' structure found in `<net/if.h>`.

SIOCCRDCTRS and SIOCRDZCTRS ioctls can be used to read or read and clear network counters. The argument to the latter two ioctls is a pointer to a counter structure ''ctrreq'' found in `<net/if.h>`.

The ioctls SIOCENABLBACK and SIOCDISABLBACK can be used to enable and disable the interface loopback mode.

Restrictions

The PUP protocol family is not supported.

Diagnostics

xna%d: port self-test failed:<*register list*>
Adapter did not pass the power-up self-test during autoconfiguration time. The port attachment fails.

xna%d: couldn't allocate...
The `xna` driver was unable to allocate memory for adapter data structures. The port attachment fails.

xna%d: port probe failed:<*register list*>
The `xna` driver was unable to bring the adapter into the initialized state. The port attachment fails.

xna%d: port init failed:<*register list*>
The `xna` driver failed to prepare the adapter for runtime use.

xna%d: port state changed, resetting:<*register list*>
The adapter issued a port state change interupt. The port state is reset.

xna%d: port reset failed
The `xna` driver was unable to bring the adapter into the initialized state during a port reset.

xna%d: command failed, error code:<*error code*>
The adapter port command failed. The *error code* gives reason for failure.

xna%d: couldn't handle af%d
A packet with an undefined protocol type has been sent to the adapter.

xna%d: addmulti failed, multicast list full
Too many multicast requests have been made.

See Also

arp(4p), inet(4f), intro(4n)

Index

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-234-1998 using a 1200- or 2400-baud modem from anywhere in the USA, Canada, or Puerto Rico. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital Subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local Digital subsidiary or approved distributor
Internal*	————	SSB Order Processing - WMO/E15 *or* Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

* For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

ULTRIX
Reference Pages Section 4: Special Files
AA-LY17B-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? ______________________________

____

____

What do you like best about this manual? ______________________________

____

____

What do you like least about this manual? ______________________________

____

____

Please list errors you have found in this manual:

Page	Description

Additional comments or suggestions to improve this manual:

____

____

____

____

What version of the software described by this manual are you using? __________

Name/Title ______________________________ Dept. __________

Company ______________________________ Date __________

Mailing Address ____

__________________ Email __________________ Phone __________

Do Not Tear – Fold Here and Tape

digital™

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST-CLASS MAIL PERMIT NO. 33 MAYNARD MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3-2/Z04
110 SPIT BROOK ROAD
NASHUA NH 03062-9987

Do Not Tear – Fold Here

Cut
Along
Dotted
Line