LynxOS Reference Manual

LynxOS Version 2.5
DOC-0174

Device Facilities
File Formats
Driver Service Calls

Printed in the United States of America.

Contents

Device Facilities

File Formats

Driver Service Calls

Device Facilities

NAME

`Intro, intro` – introduction to device facilities

DESCRIPTION

Section 4, Device Facilities, describes the facilities provided to application programs by the various system device drivers. In general, the `ioctl` system call is used to make requests of a driver. `ioctl` calls may set operational parameters, retrieve current operational parameters and status information, or somehow affect the current state of input/output queues or buffers.

`ioctl` takes three arguments. The first is an open file descriptor referencing a device. Devices are referenced by special file system entities called special `files` or `nodes` (interchangeably). `nodes` are most often kept in `/dev` or a subdirectory thereof.

The second argument to `ioctl` is an `int` which selects a particular driver function. An error is returned if the function code is unknown. The third argument is either ignored or is a pointer to a data structure appropriate to the request. For example, if a request involves retrieving an integer containing some statistic, the third argument must be a pointer to an `int`.

NETWORKING

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 may include packet fragmentation and reassembly, routing, addressing, and basic transport. A protocol-family may support multiple methods of addressing, though the current protocol implementations do not. A protocol-family is normally comprised of a number of protocols, one per `socket` type. It is not required that a protocol-family support all socket types. A protocol-family may contain multiple protocols supporting the same socket abstraction.

A protocol supports one of the socket abstractions detailed in the description of the `socket` system call. A specific protocol may 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 non-standard 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 comprise the lowest layer of the networking subsystem, interacting with the actual transport hardware. An interface may support one or more protocol families, and/or address formats.

The SYNOPSIS section of each network interface entry gives a sample specification of the related drivers for use in providing a system description.

The DIAGNOSTICS section lists messages which may appear on the console and in the system error log `/usr/adm/messages` due to errors in device operation.

Networking Using STREAMS

The TLI library functions provide an interface to the Transport Service Provider in the kernel. The Transport Service Interface is implemented in the kernel by a Streams Mapping Driver which interfaces with the BSD TCP/IP protocol implementation. This driver supports three clonable STREAMS device drivers:

```
/dev/tcp
/dev/udp
/dev/ip
```

The `/dev/tcp` device supports the `T_COTS` type of transport service; `/dev/udp` device supports the `T_CLTS` type of transport service, and `/dev/ip` device supports access to the raw IP protocol or to the ICMP protocol.

Protocols

The system currently supports only the DARPA Internet protocols fully. Consult the appropriate reference pages in this section for more information regarding the support for the protocol family.

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 */
#define AF_PUP       4  /* pup protocols: e.g. BSP */
```

Supported Ethernet Devices

LynxOS provides several Ethernet device drivers for common network interface adapters as shown in the table “Ethernet Device Drivers in LynxOS”.

Table 1: Ethernet Device Drivers in LynxOS

Platform	Device Name	Network Interface Adapter	Supporting Files
x86	wd	Western Digital and old SMC Elite ISA adapters	/sys/cfg/wd3e.cfg /sys/dheaders/if_wd3einfo.h /sys/devices/if_wd3einfo.c
x86	wd	SMC Elite, Ultra, and EtherEZ ISA Adapters	/sys/cfg/ultra.cfg /sys/dheaders/if_wd3einfo.h /sys/devices/if_wd3einfo.c
x86	el_isa	3COM Etherlink III ISA Adapter	/sys/cfg/3c509.cfg /sys/dheaders/if_3c5x9info.h /sys/devices/if_3c5x9info.c

Table 1: Ethernet Device Drivers in LynxOS

Platform	Device Name	Network Interface Adapter	Supporting Files
x86	`el_eisa`	3COM Etherlink III EISA Adapter	`/sys/cfg/3c579.cfg` `/sys/dheaders/if_3c5x9info.h` `/sys/devices/if_3c5x9info.c`
x86	`smc`	SMC EtherPower 10/100 PCI Adapter	`/sys/cfg/smc_dec21040.cfg` `/sys/dheaders/if_dec21040info.h` `/sys/devices/if_dec21040info.c`
x86 and PowerPC	`amd`	AMD79C970 PCnet PCI Singlechip Ethernet Adapter	`/sys/cfg/amd970.cfg` `/sys/dheaders/if_amd970info.h` `/sys/devices/if_amd970info.c`
PowerPC	`dec`	DEC210x4x on-board Ethernet controller	`/sys/cfg/dec21040.cfg` `/sys/dheaders/if_dec21040info.h` `/sys/devices/if_dec21040info.c`
68k	`oblan`	i82596 on-board Ethernet controller	`/sys/cfg/oblan.cfg` `/sys/dheaders/if_i82596info.h` `/sys/devices/if_i82596info.c`
68k	`hawk`	Am7990 (Lance) based VME Ethernet adapter	`/sys/cfg/hawk.cfg` `/sys/dheaders/if_hwkinfo.h` `/sys/devices/if_hwkinfo.c`
SPARC	`oblan`	Am7990 (Lance) based on-board Ethernet controller	`/sys/cfg/oblan.cfg` `/sys/dheaders/if_oblaninfo.h` `/sys/devices/if_oblaninfo.c`

The reference pages for each driver are named the same as the LynxOS device name. To see more information about the `smc` driver, for example, see `smc(4)`.

The interface name for each device, used with the `ifconfig` utility, is constructed from the device name and the unit number. For example, on a system with two SMC EtherPower 10/100 Adapters, the interface names will be `smc0` and `smc1`. See `ifconfig(4)` for more information.

NAME

`adaptec1542` – Adaptec 1540/1542 ISA SCSI device driver (x86)

DESCRIPTION

The SCSI device driver can control up to 7 devices on the SCSI bus. Such devices generally are hard disks, streaming cassette tape drives, and possibly WORM optical drives. Minor device numbers for SCSI devices are constructed from the device number. Target devices can be numbered 0 through 6; the CPU board controller is device 7. SCSI devices identify themselves and their functions to the driver, so any device can be connected without modification or reconfiguration. An exception to this rule is the TEAC MT–2ST tape drive which does not support the SCSI `INQUIRY` command. If such a device is to be used, its minor number should be adjusted by adding 128 to the device number (i.e., by setting bit 7). The driver assumes that devices with such device numbers are tape drives like the TEAC.

Features of the `adaptec1542`:

- Bus tuning parameters allow for better disk performance.

Limitations of the `adaptec1542`:

- Supports ISA DMA see below.

The 1542 host adapter is limited with respect to DMA transfers to memory. It cannot transfer to physical memory > 16 megabytes. So on machines with more then 16 megabytes of memory the driver will DMA to memory below 16 megabytes then copy the data to the proper location. This special transfer will only occur if the request contains memory > 16 megabytes. i.e. If none of the memory to be transferred to is > 16 megabytes then normal DMA is done.

The `info_structure` used by `adaptec1542` is defined as:

```
struct adapter_info {
    u_char scsi_port;      /* SCSI Bus/Port ID */
    u_char cntrl_id;       /* Controller ID*/
    u_char scsi_id;        /* SCSI ID of host adapter */
    u_short base_address;  /* Adapter Base I/O address */
    u_short dma_chan;      /* DMA channel used */
    u_short vector;    /* Interrupt Vector */
    u_short level;     /* Interrupt Level */
    u_short bus_on;    /* bus_on time (microsec) (2-15) */
    u_short bus_off;    /* bus_off time (microsec) (1-64) */
};
```

The fields in this structure are described below.

`scsi_port`	Not used.
`cntrl_id`	Not used.
`scsi_id`	ID of host adapter (should be 7).
`base_address`	Port address of host adapter.
`dma_chan`	DMA channel used by host adapter.
`level`	Not used.
`vector`	Interrupt vector used by host adapter.
`bus_on`	How long to stay on bus during DMA transfer.
`bus_off`	How long to stay off bus after DMA transfer.

The default values for `adaptec1542` are described below.

```
struct adapter_info adaptec1542_info = {
    0,                  /* Don't Care */
    0,                  /* Controller ID */
    7,                  /* SCSI ID of controller */
    0x330,              /* Controller address */
    5,                  /* DMA channel */
    (32 + 11),          /* Interrupt vector */
    0,                  /* Don't care */
    8,                  /* bus_on time (2-15) */
    40                  /* bus_off time (1-64) */
};
```

The `adaptec1542` can be tuned by setting `bus_on` and `bus_off` times in the `struct adaptec1542_info` contained in `adaptec1542_info.c`. These times are in micro-seconds. They represent the maximum amount of time the controller can stay on the bus during a DMA transfer and the minimum amount of time required to stay off the bus after performing a DMA transfer. The default settings for LynxOS are `bus_on`: 8, `bus_off`: 40. A maximum setting of `bus_on`: 15, `bus_off`: 1 can be used to boost disk performance. Setting these values can dramatically affect real time response due to DMA cycle stealing. Worst case response can double or triple. Lynx does its real time testing at the default settings.

SEE ALSO

Utility Programs - `fmtscsi`, `scandisk`

System Calls - `ioctl()`

NAME

`aio` – analog input/output card driver

SYNOPSIS

`#include <aio.h>`

`#include <ioctl.h>`

DESCRIPTION

This section describes the standard device driver used to read from analog-to-digital channels and write to digital- to-analog channels. Communication with an analog I/O board is in terms of `samples`. A single sample is generally an `int`, but this may vary depending on the actual hardware involved. Regardless of the number of bytes requested when reading from a single channel, the driver normally returns one sample (unless the number of bytes requested is less than the sample size). If timed sampling is enabled, and samples have accumulated since the last `read`, the driver will return as many as will fit in the space provided. When writing, the driver writes as many bytes as requested before returning.

The analog I/O driver can be controlled with the `ioctl` system call. The first parameter to `ioctl` should be a file descriptor referencing an appropriate special file. The third parameter should point to a structure of type `struct sgaio`, defined in **`ioctl.h`**:

```
struct sgaio {
    int tflag;              /* Control flag */
    double rate;            /* Sample rate, in seconds */
    int gain;               /* Analog input gain */
};
/* Values for tflag field */
#define  AIOTRIG   0x03    /* Mask for trigger setting */
#define  AIOETRIG  0x01    /* External trigger */
#define  AIOCTRIG  0x00    /* Request trigger  */
#define  AIOSTRIG   0x02    /* Timed trigger */
```

The second parameter to `ioctl` should be one of the following request codes, also from `ioctl.h`:

`AIOGETP`	Gets the current control settings of the driver. The settings are copied to the `sgaio` structure pointed to by the third argument.
`AIOSETP`	Configures the driver according to the values in the `sgaio` structure pointed to by the third argument.
`AIOGGRP`	Gets the channel group mask for the group including the channel referenced by the file descriptor in the first argument. The mask is written to the long integer pointed to by the third argument.
`AIOSGRP`	Sets the current channel group according to the mask in the long integer pointed to by the third argument.

Control Parameters

The triggering method, sample rate, and gain for each input or output channel can be configured. If external triggering is selected, an external source (whose nature depends on the hardware used) controls the taking of analog samples or the generation of output voltages. If timed triggering is selected, a time delay value given in `rate` dictates the frequency with which samples are taken or voltages changed. If no triggering is selected, then samples are taken or generated upon demand. Voltage gain for input and output can be controlled with `gain`.

Channel Groups

For convenience and efficiency when sampling a set of input signals or generating a set of output voltages, several channels can be grouped so that single `read` and `write` requests affect the entire set simultaneously. Up to 32 groups can be defined, and a group may contain up to 32 members. Each channel may be a member of one group at any time.

A group is described by a long integer. Each bit position in the integer represents a channel; if the bit is 1, that channel is a member of the group. Once a group is established, any channel in the group can be opened to access all channels in the group. Each `read` or `write` request will transfer one sample value for each channel in the group, arranged in order of channel number. For example, if a group contains channels 1, 2, and 6, a read request will return three samples in the buffer, the first from 1, the second from 2, and the third from 6.

Note that all control parameters and channel groupings remain intact after the file descriptors used to establish them have been closed. Often, parameters are established by the `saio` utility before starting an application. The application therefore need not concern itself with configuring the channels itself.

SEE ALSO

Utility Programs - `saio, select`

System Calls - `ioctl()`

NAME

`amd` – character device interface to Ethernet (x86, PowerPC)

DESCRIPTION

The `amd` device driver is a special character device to perform communication over Ethernet/IEEE802.3 data link. This driver supports the AM79C970A PCnet PCI II Single-chip Full-Duplex Ethernet controller.

It provides the following `ioctl` functions:

SIOCSIFADDR	Used by the `ifconfig` utility to pass-on the Internet address of the Ethernet interface.
SIOCSIFFLAGS	Used by the `ifconfig` utility to pass-on the flags to set/reset the interface.
GE_SET_DEV	Sets the interface in `raw_ethernet` mode. In this mode, the applications can directly communicate to the interface using `raw_ethernet` sockets.
GE_SET_ADDR	Changes the Ethernet address of the interface.
GE_SET_PROTO	Facilitates the setting up of the protocol range that the `raw_ethernet` socket needs to receive.
GE_SET_MCAST	Allows the user to set up a list of “multicast addresses” that the interface can receive as destination node using a `raw_ethernet` socket.
GE_PROM	When programming the interface to receive all packets, this `ioctl` utility determines whether or not it is meant for this node.
GE_DEBUG_MODE	Puts the interface in debug mode. The data transfers through this interface would be displayed in the debug terminal.
GE_GET_CTRS	Gets the `raw_ethernet` interface statistics.
GE_GET_SADDR	Gets the current Ethernet address of the interface.
GE_GET_FADDR	Gets the station address as assigned by the manufacturer of the ethernet interface hardware.

`GE_CLRIFNAME` Closes the interface for `raw_ethernet` reception for a `raw_ethernet` socket.

The `/sys(sysso)/devices/if_amd970info.c` file has the following structure for configuring the Ethernet interface:

```
struct amd970_uinfo {
    char *if_name;
    int io_mapped_device;
};
```

The fields in this structure are:

`if_name` Driver name; usually “amd”.

`io_mapped_device`
Type of device; valid only for x86 devices. When `io_mapped_device` is set to a value other than zero, all register accesses are I/O mapped. When it is set to zero, all the register accesses are memory mapped.

ERRORS

This driver may print the following messages on console/`tty` debug port:

```
buffer allocation failed !
LANCE INIT: Bad number (%x) of receive rings
LANCE INIT: Bad number (%x) of transmit rings
setup_ring: bad switch, %x
LANCE initialization failed
```

These error messages are described in the table “Troubleshooting AM79C970 Driver Error Messages”.

Table 1: Troubleshooting AM79C970 Driver Error Messages

Message	Severity	Cause	Fix
`buffer allocation failed`	Fatal	Contiguous memory is not available.	Increase the RAM in the system. If the problem happens even when 20MB RAM is available, then contact technical support.
`LANCE INIT: Bad number (%x) of receive rings`	Critical	Cable is bad or a faulty Am79C970 board.	Check the cable.
`LANCE INIT: Bad number (%x) of transmit rings`	Critical	Cable is bad or a faulty Am79C970 board.	Check the cable.
`setup_ring: bad switch, %x`	Fatal	Faulty system RAM.	Check the memory on the mother board. If the memory is OK, the cause might be some threads corrupting physical memory.
`LANCE initialization failed`	Fatal	Faulty Am79C970 board or IRQ confilicts	Check the board or the PCI configuration to see whether an IRQ slot is available to Am79C970 board.

FILES

`/sys/devices/if_amd970info.c`

`/sys/dheaders/if_amd970info.h`

`/sys/lynx.os/amd970.cfg`

`/sys/drivers/if_amd970/if_amd970.o`

`/dev/amd`

SEE ALSO

Utility Programs -	`ifconfig`
Device Facilities -	`raw_ethernet()`
Documents -	See “Raw Ethernet Support” in the LynxOS User’s Guide

NOTES

Unlike normal devices, this device cannot be operated with open/close `ioctl` calls directly. Instead, the device can be operated through sockets either in `AF_INET` domain or in `AF_RAWETH` domain.

NAME

`atc` – keyboard and console (x86)

SYNOPSIS

```
#include <kd.h>

#include <vt.h>
```

DESCRIPTION

The x86 console driver currently includes a driver for the keyboard and for monochrome, EGA, and VGA graphics boards (in text mode). Because the low-level driver uses the LynxOS terminal manager to perform all line editing functions, software designed for a normal ASCII terminal will work normally on the console. Some `ioctl` requests, of course, are meaningless, such as requests to change baud rate or to force an RS232 “break” condition; such requests are silently ignored.

Depending upon available text mode memory, the driver supports one, two, or four virtual consoles. Each is a separate device, with distinct entries in `/dev`. Normally, on a given system each available virtual console is enabled for `login` via `/etc/ttys`. Switching between the consoles can be done through `ioctl` calls or directly from the keyboard; see below. Note that the driver does not automatically determine the actual number of screens available. If attempts are made to use more virtual consoles than are meaningful given the display hardware, the extra “consoles” simply overlap each other. Some monochrome adapters allow two separate displays, which will be available as consoles 0 and 3; others only support one console. All EGA and VGA interfaces support four virtual consoles. The **Esc m** sequence (`q.v.`) can be used to enable or disable virtual consoles.

The console driver emulates a DEC VT100 terminal. In addition, it supports some of the capabilities defined by the `xterm` client of the X Window System.

Display Control

Printable characters written to the console screen are placed in screen memory and thus displayed. Each printed character advances the cursor to the right, wrapping around to the next line at the right edge and scrolling up after the last line. Note that the VT100 auto-wrap feature emulated by the `atc` driver differs from the auto-wrap features of some other terminals. For example, on a WYSE 50 terminal, after a character is written in column 80, the cursor immediately wraps around to the next line. On the `atc` driver, however, after a character is written in column 80, the cursor remains in column 80 until the next character is written to the screen, at which time the auto-wrap is performed and the new character appears in column 1 of the next line.

The `atc` driver interprets various characters and character sequences as commands to affect the current screen display. The currently recognized single-character commands are given in the table “Single Character Commands Recognized by the atc Driver”. For ease of reference, the multi-character escape sequences are divided into non-VT100 compatible and VT100 compatible sequences. The non-VT100 compatible sequences are listed in the table “Non-VT100-Compatible Escape Sequences Recognized by the atc Driver”. The VT100 compatible sequences are listed in the table “VT100-Compatible Escape Sequences Recognized by the atc Driver”, the table “More VT100-Compatible Esc Sequences Recognized by the atc Driver”, and the table “Even More VT100-Compatible Esc Sequences Recognized by the atc Driver”.

Each character is printed with display attributes determined from a value maintained by the driver. Two escape sequences (**Esc g** and **Esc h**) are available to modify the attribute byte. The following table summarizes the meaning of the attribute byte for color graphics systems.

Table 1: Single Character Commands Recognized by the `atc` Driver

ASCII code	Control character	Effect
7	**^G**	Bell. Produces an audible alarm.
8	**^H**	Backspace. No action occurs if received when cursor is at left margin.
9	**^I**	Horizontal Tab. Move to next tab stop, or to the right margin if there are no more tab stops. Tab stops initially occur at 8 character intervals, and can be modified.
10	**^J**	Linefeed. If the `linefeed/newline` feature is off, then the cursor moves to the next line but stays in the same column. If the `linefeed/newline` feature is on, then the cursor moves to the left margin of the next line.

Table 1: Single Character Commands Recognized by the `atc` Driver

ASCII code	Control character	Effect
`11`	**^K**	Vertical tab. Processed as linefeed.
`12`	**^L**	Form feed. Processed as linefeed.
`13`	**^M**	Carriage return. Cursor moves to the left margin of the current line.
`14`	**^N**	Shift out. Select **G1** character set as the current character set.
`15`	**^O**	Shift in. Select **G0** character set as the current character set.
`27`	**^[**	Escape (**Esc**). Introduce escape sequence; see below.

Table 2: Non-VT100-Compatible Escape Sequences Recognized by the `atc` Driver

Sequence	Effect
Esc <	Causes **Caps Lock** key on the keyboard to behave as an all-caps toggle switch.
Esc ?	Causes the **Caps Lock** key on the keyboard to behave as if it were another CONTROL key. The normal **Caps Lock** feature is available when **Shift** and **Caps Lock** are depressed together. This feature is useful with some improperly designed keyboards.
Esc F	Go to status line, column 34. All subsequently written printable characters will appear on the status line (line 25) beginning in column 34. The cursor will remain on the status line until a non-printable character is written.
Esc b *mode*	Set bell mode to *mode*, which must be a character between 0 and 9. Mode 0 is silent; other modes provide a variety of frequencies, from a rough buzz to a piercing high-frequency pain beam. Mode 5 is most-commonly used.

Table 2: Non-VT100-Compatible Escape Sequences Recognized by the `atc` Driver

Sequence	Effect
Esc f *addr data* ...	Begin font download sequence for EGA and VGA graphics systems. The *addr* should be a 2-byte offset from the beginning of font memory, expressed as the ASCII string representation of a 4-digit hexadecimal number. Following the address is a stream of 8-bit values expressed as the ASCII representations of two hexadecimal digits, indicating the bitmaps of the characters. The stream is terminated by any character that is not a valid hex digit.
Esc g *mask*	Set bits in current attribute byte. The *mask*, expressed as the ASCII string representation of a 2-digit hexadecimal number, is bitwise-OR'ed with the current value.
Esc h *mask*	Clear bits in current attribute byte. The complement of *mask*, expressed as the ASCII string representation of a 2-digit hexadecimal number, is bitwise-AND'ed with the current value.
Esc m *n*	Enable virtual consoles. The parameter *n* may be "1", "2", or "4".
Esc *n*	Disable the status line (line 25).
Esc t *hour min*	Set the time on the status line. Both *hour* and *min* are ASCII string representations of two-digit decimal numbers.
Esc y	Enable the status line (line 25). The status line is enabled by default.

Table 3: VT100-Compatible Escape Sequences Recognized by the `atc` Driver

Sequence	Effect
Esc (A	Designate the United Kingdom character set as **G0**.
Esc (B	Designate the United States character set as **G0**.
Esc (0	Designate the special character and line drawing character set as **G0**.
Esc) A	Designate the United Kingdom character set as **G1**.
Esc) B	Designate the United States character set as **G1**.
Esc) 0	Designate the special character and line drawing character set as **G1**.

Table 3: VT100-Compatible Escape Sequences Recognized by the `atc` Driver

Sequence	Effect
Esc 7	Save the cursor position, character attribute, character set and origin mode selection (to be restored using **Esc 8**).
Esc 8	Restore the previously saved cursor position, character attribute, character set and origin mode selection. If none were saved, the cursor moves to the home position.
Esc =	Selects alternate (application) keypad mode. In this mode, the numeric keypad transmits control functions as listed in the table “Codes Transmitted By Numeric Keypad (ASCII Values Included)”.
Esc >	Selects numeric keypad mode. In this mode, the numeric keypad transmits numeric functions as listed in the table “Codes Transmitted By Numeric Keypad (ASCII Values Included)”. This mode is the default.
Esc D	Moves the cursor down one line in the same column. If the cursor is at the bottom margin, a scroll up is performed.
Esc E	Moves the cursor to the first position on the next line. If the cursor is at the bottom margin, a scroll up is performed.
Esc H	Sets a horizontal tab stop at the cursor position.
Esc M	Moves the cursor up one line in the same column. If the cursor is at the top margin, a scroll down is performed.
Esc N	Select the United States character set for the next character (only).
Esc O	Select the special character and line drawing character set for the next character (only).
Esc c	Reset the terminal to its initial state.

Table 4: More VT100-Compatible Esc Sequences Recognized by the `atc` **Driver**

Sequence	Effect
Esc [?*p***;...;***p* **h**	This is the general form of the VT100 set sequence. Each mode specified by a *p* parameter is set. Values for *p* are listed in the table “Values For Parameters In VT100 Set and Reset Sequences”.
Esc [?*p***;...;***p* **1**	This is the general form of the VT100 reset sequence. Each mode specified by a *p* parameter is reset. Values for *p* are listed in the table “Values For Parameters In VT100 Set and Reset Sequences”.
Esc [20 h	Sets (turns on) the linefeed/newline feature. Any linefeed, form feed or vertical tab will move the cursor to the first column on the next line. The **Return** key transmits both a carriage return and linefeed.
Esc [20 l	Resets (turns off) the linefeed/newline feature. Any linefeed, form feed or vertical tab will move the cursor to the next line in the same column. The **Return** key transmits just a carriage return.
Esc [*top***;***btm* **r**	Set the scrolling region’s top and bottom margins to *top* and *btm*, respectively. Both *top* and *btm* are ASCII string representations of decimal numbers. If they are not specified, the complete screen is used (no margins).
Esc [*n* **A**	Move cursor up *n* lines in the same column, stopping at the top margin. The parameter *n* is an ASCII string representation of a decimal number. If *n* is 0 or not given, it is assumed to be 1.
Esc [*n* **B**	Move cursor down n lines in the same column, stopping at the bottom margin. The parameter n is an ASCII string representation of a decimal number. If *n* is 0 or not given, it is assumed to be 1.
Esc [*n* **C**	Move cursor right *n* columns, stopping at the right margin. The parameter *n* is an ASCII string representation of a decimal number. If *n* is 0 or not given, it is assumed to be 1.

Table 4: More VT100-Compatible Esc Sequences Recognized by the `atc` **Driver**

Sequence	Effect
Esc [*n* **D**	Move cursor left *n* columns, stopping at the left margin. The parameter *n* is an ASCII string representation of a decimal number. If *n* is 0 or not given, it is assumed to be 1.
Esc [*l*;*c* **H**	Move the cursor to line *l*, column *c*. Both *l* and *c* are ASCII string representations of decimal numbers. If either is 0 or not given, the cursor moves to the first line or column, respectively. The numbering of lines and the ability to move outside the margins depend on the origin mode.
Esc [*l*;*c* **f**	Same as **Esc [** *l*;*c* **H**.

Table 5: Even More VT100-Compatible Esc Sequences Recognized by the `atc` Driver

Sequence	Effect
Esc [*p* **m**	Selects the character attribute according to *p*, which is an ASCII representation of a decimal digit: *p* = 0 Normal *p* = 1 Bold *p* = 4 Underline *p* = 5 Blink *p* = 7 Reverse If *p* is not given, it is assumed to be 0. Note that the character attribute can also be manipulated directly using the non-VT100 sequences **Esc g** *mask* and **Esc h** *mask*. This is useful for color graphics systems. See the table "Attribute Byte For Color Graphics Systems".
Esc [g	The parameter *p* is an ASCII representation of a decimal digit. If *p* is 0 or not given, the horizontal tab stop at the cursor position is cleared. If *p* is 3, then all horizontal tab stops are cleared.
Esc [K	Erases characters from the current line according to *p*, which is an ASCII representation of a decimal digit: *p* = 0 Erase from cursor to end of line, including the cursor position. *p* = 1 Erase from the beginning of the line to the cursor, including the cursor position. *p* = 2 Erase the entire line. If *p* is not given, it is assumed to be 0.
Esc [*p* **J**	Erases characters from the display according to *p*, which is an ASCII representation of a decimal digit: *p* = 0 Erase from cursor to end of the screen, including the cursor position. *p* = 1 Erase from the beginning of the screen to the cursor, including the cursor position. *p* = 2 Erase the entire screen. The cursor does not move. If *p* is not given, it is assumed to be 0.
Esc [*p* **L**	Inserts *p* lines before the current line. The parameter *p* is an ASCII string representation of a decimal number (default: *p* = 1).

Table 5: Even More VT100-Compatible Esc Sequences Recognized by the `atc` Driver

Sequence	Effect
Esc [*p* **M**	Deletes *p* lines from the current line down. The parameter *p* is an ASCII string representation of a decimal number (default: *p*=1).
Esc [*p* **@**	Inserts *p* characters before the current character. The parameter *p* is an ASCII string representation of a decimal number (default: *p*=1).
Esc [*p* **P**	Deletes *p* characters starting at the current character. The parameter *p* is an ASCII string representation of a decimal number (default *p*=1).

Table 6: Values For Parameters In VT100 Set and Reset Sequences

Para-meter	Mode	Meaning
`1`	Arrow Mode	This mode determines the values transmitted by the arrow keys. Refer to the table “Codes Transmitted By Arrow Keys (ASCII Values Included)” for these values. The default Arrow Mode is the reset state.

Table 6: Values For Parameters In VT100 Set and Reset Sequences

Para-meter	Mode	Meaning
5	Screen Mode	If this mode is set, then the screen is reverse (black characters on white background). If it is reset, then the screen normal (white characters on black background). The screen mode is exclusive OR'ed over the entire screen, so that reverse attribute characters written to a reverse screen will appear normal (white on black). Colors are effectively complemented by reverse Screen Mode. The default screen mode is the reset state.
6	Origin Mode	When this mode is set, the cursor moves to the home position (leftmost position at the top margin), and the cursor positioning sequences cannot move the cursor out of the scrolling region. When origin mode is reset, the cursor moves to the upper left corner of the screen (possibly above the top margin), and the cursor positioning sequences can move the cursor out of the scrolling region. The default Origin Mode is the reset state.
7	Wrap Mode	If this mode is set, then any character written to the screen when the cursor is at the right margin will be displayed on the next line, scrolling the display up if necessary. If wrap mode is reset, then any character written to the screen when the cursor is at the right margin will replace the previously displayed character. The default wrap mode is the set state (auto-wrap).

Table 7: Codes Transmitted By Numeric Keypad (ASCII Values Included)

Key	Numeric Keypad Mode (default)	Application Keypad Mode
0	0 48	**Esc O p** 27 79 112
1	1 49	**Esc O q** 27 79 113
2	2 50	**Esc O r** 27 79 114

Table 7: Codes Transmitted By Numeric Keypad (ASCII Values Included)

Key	Numeric Keypad Mode (default)	Application Keypad Mode
3	3 51	**Esc O s** 27 79 115
4	4 52	**Esc O t** 27 79 116
5	5 53	**Esc O u** 27 79 117
6	6 54	**Esc O v** 27 79 118
7	7 55	**Esc O w** 27 79 119
8	8 56	**Esc O x** 27 79 120
9	9 57	**Esc O y** 27 79 121
–	– 45	**Esc O m** 27 79 109
+	+ 43	**Esc O l** 27 79 108
.	. 46	**Esc O n** 27 79 110

Table 8: Codes Transmitted By Arrow Keys (ASCII Values Included)

Key	Arrow Mode set (default)	Arrow Mode reset
	Esc [A 27 91 65	**Esc** O A 27 79 65
Ø	**Esc** [B 27 91 66	**Esc** O B 27 79 66

Table 8: Codes Transmitted By Arrow Keys (ASCII Values Included)

Key	Arrow Mode set (default)	Arrow Mode reset
¨	**Esc** [C 27 91 67	**Esc** O C 27 79 67
Æ	**Esc** [D 27 91 68	**Esc** O D 27 79 68

Table 9: Attribute Byte For Color Graphics Systems

Bit 76543210	Effect
`10000000`	Foreground blink
`01000000`	Background red on/off
`00100000`	Background green on/off
`00010000`	Background blue on/off
`00001000`	Foreground intensity
`00000100`	Foreground red on/off
`00000010`	Foreground green on/off
`00000001`	Foreground blue on/off

For monochrome adapters, the attribute byte takes on some different meanings. The blinking and intensity attributes are available, along with four different combinations of foreground/background “color”:

`00000111` White characters on a black background.

`00000001` Underlined characters (appears as blue on black on a color system).

`01110000` Black characters on a white background (colloquially known as “reverse video”).

`00000000` Invisible characters (questionably useful).

Strangely, no other plausibly interesting color values have any meaning in monochrome mode.

FONT CONTROL

The `atc` driver allows the display font to be modified, given hardware support (EGA and VGA only). The **Esc f** sequence introduces a download sequence consisting of an address and a sequence of bitmaps. Each printable glyph is allocated a 32-byte region of font memory, of which only the first 14 (EGA) or 16 (VGA) bytes are significant. Thus, to locate the bitmap of a particular character, its ASCII value should simply be multiplied by 32. For example, a sequence to change the glyph for the character “A” (ASCII code 65 decimal, 41 hex) to a filled-in 8x12 rectangle would be as follows:

```
ESC f0820ffffffffffffffffffffffff.
```

(Note that the space following the **Esc** above is a typographical artifact; no space may appear within the actual sequence.)

Font downloading is only available on virtual console 0, but the downloaded font is available to all consoles. Currently, only one font is available at any given time. When the built-in font is overwritten, it is lost.

KEYBOARD CONTROL

The `atc` driver supports all alphabetic characters on any functional PC/AT keyboard. Other supported keys are listed in the table "Special Keys Recognized by the atc Driver". The codes transmitted by the keyboard's function keys are listed in the table "Codes Transmitted By Function Keys (ASCII Values Included).".

Table 10: Special Keys Recognized by the `atc` Driver

Key	Effect
ESC	Transmits **^[**
Ctrl	While depressed, causes alphabetic keys to transmit corresponding control codes. Valid control characters whose corresponding printable characters are shifted versions of other keys are available without need of actually pressing **Shift** (i.e., **^6** is effectively **^^**).
Shift	While depressed, forces alphabetic characters to be transmitted as upper case, and makes available "uppercase" punctuation and special characters. If **Caps Lock** is in effect, forces alphabetic characters to be transmitted as lower case.
Alt	While depressed, causes all alphabetic, punctuation and keypad characters to be preceded by **Esc N**. Thus **Alt-A** transmits the three ASCII codes 27 78 65, in that order. When pressed with one of **F1**, **F2**, **F3**, or **F4**, the driver switches to virtual console 0, 1, 2, or 3. When combined with the **Shift** key, **Alt** has no effect except for switching to virtual consoles via **F1**, **F2**, **F3**, and **F4**.
Caps Lock	Toggles between uppercase "lock" and "unlock". If the string **Esc ?** has been received, **Caps Lock** is made identical to **Ctrl**.
Backspace	Transmits **^H** (ASCII code 8).
Tab	Transmits **^I** (ASCII code 9).
Return	Transmits **^J** (ASCII code 10) if linefeed/newline feature is not set (default). Transmits **^J** and **^M** (ASCII codes 10 and 13) otherwise.
Scroll Lock	If the **Scroll Lock** LED is unlit, transmits the currently defined "stop" character (usually **^S**) and lights the LED. If the LED is lit, transmits the currently defined "start" character (usually **^Q**) and extinguishes the LED.

Table 10: Special Keys Recognized by the `atc` Driver

Key	Effect
Insert	Transmits **Esc [@** (ASCII codes 27 91 64) in that order.
Delete	Transmits ASCII code 127.
Home	Transmits **Esc [H** (ASCII codes 27 91 72) in that order.
End	Transmits **Esc [O** (ASCII codes 27 91 79) in that order.
Page Up	Switches to the next virtual console in ascending sequence.
Page Down	Switches to the next virtual console in descending sequence.

Table 11: Codes Transmitted By Function Keys (ASCII Values Included).

Key	Not Shifted	Shifted
F1	**Esc O P** `27 79 80`	**Esc O p** `27 79 112`
F2	**Esc O Q** `27 79 81`	**Esc O q** `27 79 113`
F3	**Esc O R** `27 79 82`	**Esc O r** `27 79 114`
F4	**Esc O S** `27 79 83`	**Esc O s** `27 79 115`
F5	**Esc O T** `27 79 84`	**Esc O t** `27 79 116`
F6	**Esc O U** `27 79 85`	**Esc O u** `27 79 117`
F7	**Esc O V** `27 79 86`	**Esc O v** `27 79 118`
F8	**Esc O W** `27 79 87`	**Esc O w** `27 79 119`
F9	**Esc O X** `27 79 88`	**Esc O x** `27 79 120`
F10	**Esc O Y** `27 79 89`	**ESC O y** `27 79 121`

Table 11: Codes Transmitted By Function Keys (ASCII Values Included).

Key	Not Shifted	Shifted
F11	**Esc O Z** 27 79 90	**Esc O z** 27 79 122
F12	**Esc O A** 27 79 65	**Esc O a** 27 79 97

Because so many keyboards were designed under the impression that the ability to lock the keyboard into uppercase mode is more important than the ability to transmit control codes, the driver is capable of treating the **Caps Lock** key as an extra **Control** key. If the escape sequence, **Esc ?**, is written to the console, the **Caps Lock** key becomes a **Control** key; the real **Control** key is unaffected. The ordinary uppercase lock feature is available with the combination of **Shift** and **Caps Lock**. The escape sequence, **Esc <**, forces the **Caps Lock** key to resume its normal role. In either mode, when uppercase lock is in effect the **Caps Lock** LED on the keyboard is lit.

If the **Control**, **Shift**, and **Caps Lock** keys are pressed together, the driver enters a “monitor” mode. No screen control commands are recognized; all characters received are displayed on the screen. Normal printable characters are displayed unmodified. Control characters (those with codes between 0 and 31) are displayed as the corresponding printable character in reverse video. Code 127 is displayed as a reverse video question mark. If the eighth bit of a character is on, the character is displayed as above, but blinking. When monitor mode is enabled, the **Num Lock** LED is lit. The key sequence used to enter monitor mode is used to exit monitor mode as well.

IOCTL CALLS

The following `ioctl` calls are handled by the `atc` driver. All others are passed on to the LynxOS terminal manager.

`TIO_ENSCANMODE`
: Enable scan mode.

`TIO_DISSCANMODE`
: Disable scan mode.

`TIOSYNCEN` Enable vertical sync interrupt.

TIOWSYNC	Wait for vertical sync interrupt.
TIOCGWINSZ	Return the parameters for the dimensions of the screen in the `winsize` structure pointed to by the argument. The `winsize` structure is defined in `<ioctl.h>`.
VT_LOCKDISP	Disable the ability to switch virtual consoles from the keyboard using **Page Up**, **Page Down**, or any of the **Alt-F*** combinations.
VT_UNLOCKDISP	Enable the ability to switch virtual consoles from the keyboard using **Page Up**, **Page Down**, or any of the **Alt-F*** combinations.
VT_OPENQRY	Get an available virtual console and return its number in the long pointed to by the argument. The available virtual console that is returned is the lowest number virtual console that is not opened by any process. If there is no available virtual console, the `ioctl` call will return `-1` and set **errno** to `ENXIO`.
VT_GETMODE	Get the mode of the virtual console and return it in the `vt_mode` structure pointed to by the argument. The `vt_mode` structure is defined in `<vt.h>`.
VT_SETMODE	Set the mode of the virtual console according to the `vt_mode` structure pointed to by the argument. The `vt_mode` structure is defined in `<vt.h>`.
VT_RELDISP	Tells the `atc` driver that the virtual console has or has not been released by the process. A non-zero argument indicates that the virtual console has been released, and a zero argument indicates that it has not.
VT_ACTIVATE	Activate the virtual console specified by the given argument. The `atc` driver switches to the new console just as if an **Alt-F*** combination or **Page Up** or **Page Down** had been transmitted by the keyboard. If the specified virtual console is not open or does not exist, the `ioctl` call will return `-1` and set **errno** to `ENXIO`.
VT_GETSTATE	Return the console state information in the `vt_stat` structure pointed to by the argument. The `vt_stat` structure is defined in `<vt.h>`. The active virtual

	terminal is stored in the `v_active` field. The `v_state` field is a bit mask such that bit `X` is set if and only if the virtual console `X` is open.
`VT_SENDSIG`	Using the `vt_stat` structure pointed to by the argument, send the signal specified by the `v_signal` field to those virtual consoles specified by the `v_state` bit mask field. The `vt_stat` structure is defined in `<vt.h>`.
`VT_KBGET`	Return the current key translation tables in the `remote_loadable_atran_t` pointed to by the argument. The `remote_loadable_atran_t` type is defined in `<vt.h>`.
`VT_KBPUT`	Set the key translation tables according to the contents of the `remote_loadable_atran_t` pointed to by the argument. The `remote_loadable_atran_t` type is defined in `<vt.h>`.
`VT_KBDFT`	Reset the key translation tables to the default.
`KIOCSOUND`	Start or stop sound generation. A non-zero argument means start sound generation at the frequency specified by the argument. A zero argument means stop sound generation.
`KDMKTONE`	Generate a tone of fixed frequency and duration. The lower 16 bits of the argument indicate the frequency, and the upper 16 bits indicate the duration in milliseconds.

SEE ALSO

Utility Programs -	`tset, stty`
System Calls -	`ioctl()`
Device Facilities -	`tty`

NAME

`atc` – international extended `atc` driver (x86)

SYNOPSIS

`#include <vt.h>`

DESCRIPTION

The `atc` console driver shipped with LynxOS for x86 supports the standard 101-keys US-type keyboard only. The international extended `atc` driver was designed to support the various national variants of the x86 keyboard including both the 101-keys US version and the 102-keys international version.

The important points are:

1. The driver supports full 8-bit codes for output (`0x01..0xff`) where the character codes conform to the ISO Latin-1 Supplementary Characterset, as far as this is possible using the IBM Set #2 (see below).
2. Accents etc. are supported on the keyboard (compose feature)
3. The righthand **Alt** key can be used as an additional meta-key
4. The driver's Keyboard Translation Tables (KKT) can be changed in runtime
5. A special utility named `keyb` (see `keyb`) is provided which builds the KTTs from a user-definable textfile (see `keyb`) and then passes the new KTTs to the driver
6. A special key combination, **Ctrl-Alt-Esc**, is provided which forces the driver to reset the KTTs to their defaults (standard 101-keys US type keyboard)
7. The international extended `atc` driver preserves compatibility with the standard version of the `atc` driver
8. There are 3 new `ioctl()` calls: `VT_KBDGET`, `VB_KBDPUT`, and `VB_KBDDFT` which are used for reading, writing and resetting the internal KTTs, respectively. Customer software need not call these functions.

The Extended Output Characterset

Since the International Extended `atc` Driver is now able to display codes in the range `0x01..0xff` there are some things to know about the characters attached to the new codes. These are shown in the table "Characters in Ranges 0x01 to 0xff"

Table 1: Characters in Ranges 0x01 to 0xff

Code Range	Description
0x01..0x1f	Most of these codes are passed unchanged to the character display so they show the IBM additional character set which comprises some funny symbols. Some codes (e.g., CR, LF, VT, BEL) are non-printing since they are used as control codes (C0 set) for the ANSI terminal standard. 0x00 is always a no-op code (non-printing and no control effects).
0x20..0x7e	These are the "usual" characters, absolutely unchanged with respect to earlier versions of the atc driver, including the switching of the Special Graphics Characters.
0x7f..0xbf	These codes are passed unchanged to the display driver so they will show what the IBM Character Set #2 provides for this range. Please note that these codes are used as control codes (C1 set) for some ANSI terminals in 8-bit mode.
0xa0..0xff	This is the range of the "ISO Latin-1" Supplementary Character set, as far as it could be done on base of the IBM Character set #2. Symbols which are not provided in the IBM set will appear as a double exclamation mark. The "ISO Latin-1" set may be used with the keyb utility when defining national keyboard layouts.

Since the standard manual browser software (`troff, man`) is unable to handle character codes outside the range `0x20..0x7f` the character table was not included in this document. The table is provided in file `atc-4X-chartab` (directory `/usr/local/doc`). Once the International Extended `atc` Driver has been installed `atc-4X-chartab` can be shown using `more` or `cat`.

FILE

`atc-4X-chartab`

SEE ALSO

Utility Programs - `keyb`

File Format - `keyb`

NAME

`bdlan` – Pseudo BSD network interface

SYNOPSIS

#include <ioctl.h>

DESCRIPTION

The `bdlan` Pseudo BSD network interface is part of the Hybrid Ethernet Interface (see `hybrid_ethernet(4)`) facility for the on-board Ethernet interface based on the LANCE chip on the SPARC. It supports the SMC EtherPower on the x86 and the on-board DEC21040 Ethernet controller on the PowerPC platform. Using the `dlbsd` utility it can be attached to the corresponding STREAMS-based Ethernet driver for transmission/reception of the Internet Protocol packets. The `bdlan` interface provides the following `ioctl` functions:

`SIOCSIFADDR`	Used by the `ifconfig` utility to pass-on the internet address of the Ethernet interface.
`SIOCSIFFLAGS`	Used by the `ifconfig` utility to pass-on the flags to set/reset the interface.

Currently, it does not support the raw Ethernet interface that is normally supported by the regular BSD network interface drivers for LynxOS. But an equivalent interface is provided by the STREAMS-based Ethernet driver `dlpilan` on the SPARC.

SEE ALSO

Utility Programs -	`dlbsd, ifconfig`
Device Facilities -	`Hybrid Ethernet Interface, dlpilan, dlpi_21040`

NAME

`bpf –` Berkeley Packet Filter

SYNOPSIS

`pseudo-device bpfilter 16`

DESCRIPTION

The Berkeley Packet Filter provides a raw interface to data link layers in a protocol independent fashion. All packets on the network, even those destined for other hosts, are accessible through this mechanism.

The packet filter appears as a character special device, `/dev/bpf0`, `/dev/bpf1`, etc. After opening the device, the file descriptor must be bound to a specific network interface with the `BIOSETIF ioctl`.

A given interface can be shared be multiple listeners, and the filter underlying each descriptor will see an identical packet stream. The total number of open files is limited to the value given in the kernel configuration; the example given in the SYNOPSIS above sets the limit to 16.

A separate device file is required for each minor device. If a file is in use, the open will fail and errno will be set to `EBUSY`.

Associated with each open instance of a bpf file is a user-settable packet filter. Whenever a packet is received by an interface, all file descriptors listening on that interface apply their filter. Each descriptor that accepts the packet receives its own copy.

Reads from these files return the next group of packets that have matched the filter. To improve performance, the buffer passed to read must be the same size as the buffers used internally by `bpf`. This size is returned by the `BIOCGBLEN ioctl` (see below), and under BSD, can be set with `BIOCSBLEN`. Note that an individual packet larger than this size is necessarily truncated.

The packet filter will support any link level protocol that has fixed length headers. Currently, only Ethernet, SLIP and PPP drivers have been modified to interact with `bpf`.

Since packet data is in network byte order, applications should use the htonl(3) macros to extract multi-byte values.

A packet can be sent out on the network by writing to a bpf file descriptor. The writes are unbuffered, meaning only one packet can be processed per write. Currently, only writes to Ethernets and SLIP links are supported.

IOCTLS

The ioctl command codes below are defined in <net/bpf.h>. All commands require these includes:

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/ioctl.h>
#include <net/bpf.h>
```

Additionally, BIOCGETIF and BIOCSETIF require <net/if.h>.

In addition to FIONREAD and SIOCGIFADDR, the following commands may be applied to any open bpf file. (SIOCGIFADDR is obsolete under BSD systems.)

SIOCGIFCONF should be used to query link-level addresses.) The (third) argument to the ioctl should be a pointer to the type indicated.

BIOCGBLEN(u_int)
: Returns the required buffer length for reads on bpf files.

BIOCSBLEN(u_int)
: Sets the buffer length for reads on bpf files. The buffer must be set before the file is attached to an interface with BIOCSETIF. If the requested buffer size cannot be accommodated, the closest allowable size will be set and returned in the argument. A read call will result in EIO if it is passed a buffer that is not this size.

BIOCGDLT(u_int)
: Returns the type of the data link layer underlying the attached interface. EINVAL is returned if no interface has been specified. The device types, prefixed with “DLT_”, are defined in <net/bpf.h>.

`BIOCPROMISC` Forces the interface into promiscuous mode. All packets, not just those destined for the local host, are processed. Since more than one file can be listening on a given interface, a listener that opened its interface non- promiscuously may receive packets promiscuously. This problem can be remedied with an appropriate filter.

The interface remains in promiscuous mode until all files listening promiscuously are closed.

`BIOCFLUSH` Flushes the buffer of incoming packets, and resets the statistics that are returned by `BIOCGSTATS`.

`BIOCGETIF(struct ifreq)`
Returns the name of the hardware interface that the file is listening on. The name is returned in the `if_name` field of `ifr`. All other fields are undefined.

`BIOCSETIF(struct ifreq)`
Sets the hardware interface associate with the file. This command must be performed before any packets can be read. The device is indicated by name using the `if_name` field of the `ifreq`. Additionally, performs the actions of `BIOCFLUSH`.

`BIOCSRTIMEOUT`
`BIOCGRTIMEOUT(struct timeval)`
Set or get the read timeout parameter. The `timeval` specifies the length of time to wait before timing out on a read request. This parameter is initialized to zero by `open`(2), indicating no timeout.

`BIOCGSTATS(struct bpf_stat)`
Returns the following structure of packet statistics:

```
struct bpf_stat {
 u_int bs_recv;
 u_int bs_drop; };
```

The fields are:

`bs_recv` the number of packets received by the descriptor since opened or reset (including any buffered since the last read call); and `bs_drop` the number of packets which were accepted by the filter but dropped by the kernel because of buffer overflows (i.e., the application's reads aren't keeping up with the packet traffic).

`BIOCIMMEDIATE(u_int)`

Enable or disable "immediate mode", based on the truth value of the argument. When immediate mode is enabled, reads return immediately upon packet reception. Otherwise, a read will block until either the kernel buffer becomes full or a timeout occurs. This is useful for programs like `rarpd`(1), which must respond to messages in real time. The default for a new file is off.

`BIOCSETF(struct bpf_program)`

Sets the filter program used by the kernel to discard uninteresting packets. An array of instructions and its length is passed in using the following structure:

```
struct bpf_program {
 int bf_len;
 struct bpf_insn *bf_insns; };
```

The filter program is pointed to by the `bf_insns` field while its length in units of '`struct bpf_insn`' is given by the `bf_len` field. Also, the actions of `BIOCFLUSH` are performed.

See section FILTER MACHINE for an explanation of the filter language.

`BIOCVERSION(struct bpf_version)`

Returns the major and minor version numbers of the filter language currently recognized by the kernel. Before installing a filter, applications must check that the current version is compatible with the running kernel. Version numbers are compatible if the major numbers match and the application minor is less than or equal to the kernel minor. The kernel version number is returned in the following structure:

```
struct bpf_version {
 u_short bv_major;
 u_short bv_minor; };
```

The current version numbers are given by `BPF_MAJOR_VERSION` and `BPF_MINOR_VERSION` from `<net/bpf.h>`. An incompatible filter may result in undefined behavior (most likely, an error returned by `ioctl()` or haphazard packet matching).

BPF HEADER

The following structure is prepended to each packet returned by `read`:

```
struct bpf_hdr {
 struct timeval bh_tstamp;
 u_long bh_caplen;
 u_long bh_datalen;
 u_short bh_hdrlen; };
```

The fields, whose values are stored in host order, and are:

`bh_tstamp` The time at which the packet was processed by the packet filter.

`bh_caplen` The length of the captured portion of the packet. This is the minimum of the truncation amount specified by the filter and the length of the packet.

`bh_datalen` The length of the packet off the wire. This value is independent of the truncation amount specified by the filter.

`bh_hdrlen` The length of the BPF header, which may not be equal to `sizeof(struct bpf_hdr)`.

The `bh_hdrlen` field exists to account for padding between the header and the link level protocol. The purpose here is to guarantee proper alignment of the packet data structures, which is required on alignment sensitive architectures and improves performance on many other architectures. The packet filter insures that the `bpf_hdr` and the network layer header will be word aligned. Suitable precautions must be taken when accessing the link layer protocol fields on alignment restricted machines. (This isn't a problem on an Ethernet, since the type field is a short falling on an even offset, and the addresses are probably accessed in a bytewise fashion).

Additionally, individual packets are padded so that each starts on a word boundary. This requires that an application has some knowledge of how to get from packet to packet. The macro `BPF_WORDALIGN` is defined in `<net/bpf.h>` to facilitate this process. It rounds up its argument to the nearest word aligned value (where a word is `BPF_ALIGNMENT` bytes wide).

For example, if ‘`p`’ points to the start of a packet, this expression will advance it to the next packet:

```
p = (char *)p + BPF_WORDALIGN(p->bh_hdrlen + p->bh_caplen)
```

For the alignment mechanisms to work properly, the buffer passed to `read()` must itself be word aligned.

`malloc()` will always return an aligned buffer.

FILTER MACHINE

A filter program is an array of instructions, with all branches forwardly directed, terminated by a return instruction. Each instruction performs some action on the pseudo-machine state, which consists of an accumulator, index register, scratch memory store, and implicit program counter.

The following structure defines the instruction format:

```
struct bpf_insn {
 u_short code;
 u_char jt;
 u_char jf;
 long k; };
```

The `k` field is used in different ways by different instructions, and the `jt` and `jf` fields are used as offsets by the branch instructions. The opcodes are encoded in a semi-hierarchical fashion. There are eight classes of instructions: `BPF_LD`, `BPF_LDX`, `BPF_ST`, `BPF_STX`, `BPF_ALU`, `BPF_JMP`, `BPF_RET`, and `BPF_MISC`. Various other mode and operator bits are OR-operated upon to merge them into the class that can give the actual instructions. The classes and modes are defined in `<net/bpf.h>`.

Below are the semantics for each defined BPF instruction. We use the convention that A is the accumulator, X is the index register, P[] packet data, and M[] scratch memory store. P[i:n] gives the data at byte offset "i" in the packet, interpreted as a word (n=4), unsigned halfword (n=2), or unsigned byte (n=1). M[i] gives the i'th word in the scratch memory store, which is only addressed in word units. The memory store is indexed from 0 to `BPF_MEMWORDS-1`. The fields `k`, `jt`, and `jf` are the corresponding fields in the instruction definition. "`len`" refers to the length of the packet.

`BPF_LD` These instructions copy a value into the accumulator. The type of the source operand is specified by an "addressing mode" and can be a constant (`BPF_IMM`), packet data at a fixed offset (`BPF_ABS`), packet data at a variable offset (`BPF_IND`), the packet length (`BPF_LEN`), or a word in the scratch memory store (`BPF_MEM`). For `BPF_IND` and `BPF_ABS`, the data size must be specified as a word (`BPF_W`), halfword (`BPF_H`), or byte (`BPF_B`). The semantics of all the recognized `BPF_LD` instructions follow.

`BPF_LD+BPF_W+BPF_ABS`
A <- P[k:4]

`BPF_LD+BPF_H+BPF_ABS`
A <- P[k:2]

`BPF_LD+BPF_B+BPF_ABS`
A <- P[k:1]

`BPF_LD+BPF_W+BPF_IND`
A <- P[X+k:4]

`BPF_LD+BPF_H+BPF_IND`
A <- P[X+k:2]

`BPF_LD+BPF_B+BPF_IND`
A <- P[X+k:1]

`BPF_LD+BPF_W+BPF_LEN`
A <- len

`BPF_LD+BPF_IMM`
A <- k

`BPF_LD+BPF_MEM`
A <- M[k]

`BPF_LDX` These instructions load a value into the index register. Note that the addressing modes are more restricted than those of the accumulator loads, but they include `BPF_MSH`, a hack for efficiently loading the IP header length.

`BPF_LDX+BPF_W+BPF_IMM`
X <- k

`BPF_LDX+BPF_W+BPF_MEM`
X <- M[k]

`BPF_LDX+BPF_W+BPF_LEN`
X <- len

`BPF_LDX+BPF_B+BPF_MSH`
X <- 4*(P[k:1]&0xf)

`BPF_ST` This instruction stores the accumulator into the scratch memory. We do not need an addressing mode since there is only one possibility for the destination.

`BPF_ST` M[k] <- A

`BPF_STX` This instruction stores the index register in the scratch memory store.

`BPF_STX` M[k] <- X

`BPF_ALU` The ALU instructions perform operations between the accumulator and index register or constant, and store the result back in the accumulator. For binary operations, a source mode is required (`BPF_K` or `BPF_X`).

`BPF_ALU+BPF_ADD+BPF_K`
A <- A + k

`BPF_ALU+BPF_SUB+BPF_K`
A <- A – k

`BPF_ALU+BPF_MUL+BPF_K`
A <- A * k

`BPF_ALU+BPF_DIV+BPF_K`
A <- A / k

`BPF_ALU+BPF_AND+BPF_K`
A <- A & k

BPF_ALU+BPF_OR+BPF_K
A <- A | k

BPF_ALU+BPF_LSH+BPF_K
A <- A << k

BPF_ALU+BPF_RSH+BPF_K
A <- A >> k

BPF_ALU+BPF_ADD+BPF_X
A <- A + X

BPF_ALU+BPF_SUB+BPF_X
A <- A – X

BPF_ALU+BPF_MUL+BPF_X
A <- A * X

BPF_ALU+BPF_DIV+BPF_X
A <- A / X

BPF_ALU+BPF_AND+BPF_X
A <- A & X

BPF_ALU+BPF_OR+BPF_X
A <- A | X

BPF_ALU+BPF_LSH+BPF_X
A <- A << X

BPF_ALU+BPF_RSH+BPF_X
A <- A >> X

BPF_ALU+BPF_NEG
A <--A

BPF_JMP

The jump instructions alter flow of control. Conditional jumps compare the accumulator against a constant (BPF_K) or the index register (BPF_X). If the result is true (or non-zero), the true branch is taken, otherwise the false branch is taken. Jump offsets are encoded in 8 bits so the longest jump is 256 instructions. However, the jump always (BPF_JA) opcode uses the 32 bit k field as the offset, allowing arbitrarily distant destinations. All conditionals use unsigned comparison conventions.

BPF_JMP+BPF_JA
pc += k

`BPF_JMP+BPF_JGT+BPF_K`
pc += (A > k) ? jt : jf

`BPF_JMP+BPF_JGE+BPF_K`
pc += (A >= k) ? jt : jf

`BPF_JMP+BPF_JEQ+BPF_K`
pc += (A == k) ? jt : jf

`BPF_JMP+BPF_JSET+BPF_K`
pc += (A & k) ? jt : jf

`BPF_JMP+BPF_JGT+BPF_X`
pc += (A > X) ? jt : jf

`BPF_JMP+BPF_JGE+BPF_X`
pc += (A >= X) ? jt : jf

`BPF_JMP+BPF_JEQ+BPF_X`
pc += (A == X) ? jt : jf

`BPF_JMP+BPF_JSET+BPF_X`
pc += (A & X) ? jt : jf

`BPF_RET` The return instructions terminate the filter program and specify the amount of packet to accept (i.e., they return the truncation amount). A return value of zero indicates that the packet should be ignored. The return value is either a constant (`BPF_K`) or the accumulator (`BPF_A`).

`BPF_RET+BPF_A`
accept A bytes

`BPF_RET+BPF_K`
accept k bytes

`BPF_MISC` The miscellaneous category was created for anything that doesn't fit into the above classes, and for any new instructions that might need to be added. Currently, these are the register transfer instructions that copy the index register to the accumulator or vice versa.

`BPF_MISC+BPF_TAX`
X <- A

`BPF_MISC+BPF_TXA`
A <- X

The BPF interface provides the following macros to facilitate array initializers:

`BPF_STMT(`*opcode*, *operand*`)`

and

`BPF_JUMP(`*opcode*, *operand* , *true_offset*, *false_offset*`)`

EXAMPLES

The following filter is taken from the Reverse ARP Daemon. It accepts only Reverse ARP requests.

```
struct bpf_insn insns[] = {
 BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 12),
 BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, ETHERTYPE_REVARP, 0, 3),
 BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 20),
 BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, REVARP_REQUEST, 0, 1),
 BPF_STMT(BPF_RET+BPF_K, sizeof(struct ether_arp) +
 sizeof(struct ether_header)),
 BPF_STMT(BPF_RET+BPF_K, 0), };
```

This filter accepts only IP packets between host `128.3.112.15 and 128.3.112.35`.

```
struct bpf_insn insns[] = {
 BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 12),
 BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, ETHERTYPE_IP, 0, 8),
 BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 26),
 BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x8003700f, 0, 2),
 BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 30),
 BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x80037023, 3, 4),
 BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x80037023, 0, 3),
 BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 30),
 BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x8003700f, 0, 1),
 BPF_STMT(BPF_RET+BPF_K, (u_int)-1),
 BPF_STMT(BPF_RET+BPF_K, 0), };
```

Finally, this filter returns only TCP finger packets. We must parse the IP header to reach the TCP header. The `BPF_JSET` instruction checks that the IP fragment offset is 0 so we are sure that we have a TCP header.

```
struct bpf_insn insns[] = {
 BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 12),
 BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, ETHERTYPE_IP, 0, 10),
 BPF_STMT(BPF_LD+BPF_B+BPF_ABS, 23),
 BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, IPPROTO_TCP, 0, 8),
 BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 20),
 BPF_JUMP(BPF_JMP+BPF_JSET+BPF_K, 0x1fff, 6, 0),
 BPF_STMT(BPF_LDX+BPF_B+BPF_MSH, 14),
 BPF_STMT(BPF_LD+BPF_H+BPF_IND, 14),
 BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 79, 2, 0),
 BPF_STMT(BPF_LD+BPF_H+BPF_IND, 16),
 BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 79, 0, 1),
 BPF_STMT(BPF_RET+BPF_K, (u_int)-1),
 BPF_STMT(BPF_RET+BPF_K, 0), };
```

SEE ALSO

Utility Programs - `tcpdump`

Articles - McCanne, S. and Jacobson V. The BSD Packet Filter: A New Architecture for User-level Packet Capture. *Proceedings of the 1993 Winter USENIX Technical Conference*, San Diego, CA.

FILES

`/dev/bpf*`

BUGS

The read buffer must be of a fixed size (returned by the `BIOCGBLEN ioctl`).

A file that does not request promiscuous mode may receive promiscuously received packets as a side effect of another file requesting this mode on the same hardware interface. This could be fixed in the kernel with additional processing overhead. However, we favor the model where all files must assume that the interface is promiscuous, and if so desired, must utilize a filter to reject foreign packets.

Data link protocols with variable length headers are not currently supported.

Under SunOS, if a BPF application reads more than 2^31 bytes of data, read will fail in `EINVAL`. You can either fix the bug in SunOS, or `lseek` to 0 when read fails for this reason.

“Immediate mode” and the “read timeout” are misguided features. This functionality can be emulated with non- blocking mode and `select()`.

HISTORY

The Enet packet filter was created in 1980 by Mike Accetta and Rick Rashid at Carnegie-Mellon University. Jeffrey Mogul, at Stanford, ported the code to BSD and continued its development from 1983 on. Since then, it has evolved into the Ultrix Packet Filter at DEC, a STREAMS NIT module under SunOS 4.1, and DEC.

AUTHORS

Steven McCanne, of Lawrence Berkeley Laboratory, implemented BPF in Summer 1990. The design was in collaboration with Van Jacobson, also of Lawrence Berkeley Laboratory.

NAME

`bplane` – backplane driver for MVME MPU module (PowerPC, 68K)

DESCRIPTION

The `bplane` device driver is a special character device to perform inter-CPU communication between multiple MVME modules over them VME bus. Currently this support is available for MVME167/162/177 and MVME1600 Series CPU modules. This device driver provides the following `ioctl` functions:

`SIOCSIFADDR` Used by the `ifconfig` utility to setup the IP address for the VME interface.

`SIOCSIFFLAGS` Used by the `ifconfig` utility to setup the flags to set/reset the interface

`SIOCCONFIGBKP` Used by the `bpconfig` utility to setup the configuration parameters for the MVME modules sharing the VME bus

`SIOCGETCONFIGBKP`
Used by the `bpconfig` utility to read the current configuration parameters.

The `/sys/devices/if_bplaneinfo.c` file has the following structure that can be used to configure the parameters for the VME interface communication.

```
struct bplane_uinfo  {
    short basevector ;     /* VME interrupt base vector */
    short mtusize ;       /* Max MTU size for data transfer
                              over VME */
    short maxnoproc ;     /* Max num of VME modules
                              currently configured */
} ;
```

FILES

`/sys/devices/if_bplaneinfo.c`

`/dev/bplane`

SEE ALSO

Utility Programs - `bpconfig(), ifconfig()`

File Formats - `bplane.conf()`

NOTES

This utility is used only if multiple MVME MPU VME modules are configured in an SCMP fashion over the VME bus.

NAME

`cdrom` – reference page for CD-ROM support

DESCRIPTION

LynxOS supports the ISO9660 file system and the ISO9660 + Rock Ridge Interchange Protocol file systems. LynxOS provides a utility called `mkisofs` to make an ISO file system from input files. These files can be written to blank CD-ROM by a CD Recorder (CD-R) with the utility called `cdwrite`. This CD can then be mounted and processed as if it were a read-only file system. LynxOS supports CD-ROM on all platforms and all devices. On IDE CD-ROM support is achieved with AT Atachment Packet Interface (ATAPI). CD-ROM support is seemlessly integrated into LynxOS.

When mounting a CD-ROM, specify the block device with minor device 0. For example:

```
mount /dev/sd1 /mnt -r

mount /dev/sd2940.4 /mnt -r
```

All utilities will work properly on the mounted device.

Currently, LynxOS supports booting from CD-ROMs on PowerPC, 68k and SPARC platforms. Not all BIOS's for x86 support directly booting CD-ROM so a boot floppy to boot preboot is provided which can then boot a kernel on an ISO9660 file system. The way to make a CD-ROM bootable is by running makeboot on the file produced by mkisofs. This file will be modified in a platform dependent way so it will boot a file specified with the -b option from mkisofs.

`getdents()` has been improved so it returns data in a file system independent way. This being the case readdir has been improved to call getdents instead of interpreting a LynxOS Fast Lynx File System (FLFS) only. All programs using the pre-2.5.0 `readdir` must be relinked to get ISO9660 file system capabilities. Also since `getdents` is new to 2.5.0 (Non-System V version) utilities linked with the 2.5.0 `readdir` will not run on older versions of LynxOS.

SEE ALSO

Utility Programs - `mkisofs`, `cdwrite`, `mount`

Library Functions - `getdents()`

Device Facilities - `ide`, `iso9660`

NAME

`clone` – STREAMS clone driver

DESCRIPTION

`clone` is a STREAMS software driver which is used to open an unused minor device on another STREAMS driver. These other STREAMS drivers are referred to as clonable device drivers and are configured as different minor devices with the same major device number as the clone device. When a clonable device is opened, the clone driver interprets the minor device number as the major device number of a STREAMS driver and calls the `open` function for that driver with a flag indicating cloning. This `open` function will open an unused minor device and return this minor device number to the clone `open` function. The clone `open` function then performs all the necessary operations so that subsequent system calls on that open device work without involving the clone driver.

FILES

`/dev/clone`

SEE ALSO

Device Facilities - `spx`

File Formats - `config`

NAME

dlpi_dec – STREAMS-based driver Ethernet interface (x86, PowerPC)

SYNOPSIS

#include <dlpi.h>

#include <dlpiuser.h>

DESCRIPTION

The dlpi_dec is the STREAMS-based driver for the DEC21040/SMC EtherPower Ethernet interface. This driver is part of the hybrid_ethernet facility. It implements the STREAMS-based DLPI (Data Link Provider Interface) specification and also provides support for attachment to the Pseudo BSD Network Interface bdlan to support the LynxOS TCP/IP implementation.

See the description of the device facility hybrid_ethernet for more details on the facilities provided by the STREAMS-based Ethernet Driver.

FILES

/sys/devices/dlpi_21040info.c

/dev/dlpi_dec

SEE ALSO

Utility Programs - dlbsd

Device Facilities - hybrid_ethernet, bdlan

NAME

`dlpilan` – STREAMS-based driver for on-board Ethernet (SPARC)

SYNOPSIS

`#include <dlpi.h>`

`#include <dlpiuser.h>`

DESCRIPTION

For SPARC systems only.

The `dlpilan` is the STREAMS-based driver for the on-board Ethernet interface which uses the LANCE chip. This driver is part of the `Hybrid Ethernet Interface` facility. It implements the STREAMS-based DLPI (Data Link Provider Interface) specification and also provides support for attachment to the Pseudo BSD Network Interface `bdlan` to support the LynxOS TCP/IP implementation.

See the description of the device facility `hybrid_ethernet` for more details on the facilities provided by the STREAMS-based Ethernet Driver.

FILES

`/dev/dlpilan`

SEE ALSO

Utility Programs - `dlbsd`

Device Facilities - `hybrid_ethernet`, `bdlan`

NAME

el_eisa – interface to Ethernet (x86)

DESCRIPTION

The el_eisa device driver is a special character device to perform communication over Ethernet/IEEE802.3 data link. This driver supports the 3COM EtherLink III 3C579 Ethernet adapter for ISA/EISA Bus.

It provides the following ioctl functions:

SIOCSIFADDR	Used by the ifconfig utility to pass-on the Internet address of the Ethernet interface.
SIOCSIFFLAGS	Used by the ifconfig utility to pass-on the flags to set/reset the interface.
GE_SET_DEV	Sets the interface in raw_ethernet mode. In this mode, the applications can directly communicate to the interface using raw_ethernet sockets.
GE_SET_ADDR	Changes the Ethernet address of the interface.
GE_SET_PROTO	Facilitates the setting up of the protocol range that the raw_ethernet socket needs to receive.
GE_SET_MCAST	Allows the user to set up a list of “multicast addresses” that the interface can receive as destination node using a raw_ethernet socket.
GE_PROM	When programming the interface to receive all packets, this ioctl utility determines whether or not it is meant for this node.
GE_DEBUG_MODE	Puts the interface in debug mode. The data transfers through this interface would be displayed in the debug terminal.
GE_GET_CTRS	Gets the raw_ethernet interface statistics.
GE_GET_SADDR	Gets the current Ethernet address of the interface.
GE_GET_FADDR	Gets the station address as assigned by the manufacturer of the ethernet interface hardware.

GE_CLRIFNAME Closes the interface for raw_ethernet reception for a raw_ethernet socket.

The /sys/devices/if_3c579info.c file has the following structure for configuring the Ethernet interface:

```
struct if_3c579_info {
    long iobase;    /* io base address ISA:
                       0x200-0x3E0, EISA: 0x3F0 */
    long vector;    /* 32 + IRQ  */
    int slot;       /* EISA slot number      */
    int xcvr_type;  /* TP=0, AUI=1, BNC=3    */
};
```

FILES

/sys/devices/if_3c579info.c

/dev/el_eisa

SEE ALSO

Utility Programs - ifconfig

Device Facilities - raw_ethernet

Documents - See “Raw Ethernet Support” in the LynxOS User’s Guide

NOTES

Unlike normal devices, this device cannot be operated with open/close ioctl calls directly. Instead, the device can be operated through sockets either in AF_INET domain or in AF_RAWETH domain.

NAME

`el_isa` – interface to Ethernet (x86)

DESCRIPTION

The `el_isa` device driver is a special character device to perform communication over an Ethernet/IEEE802.3 data link. This driver supports the 3COM EtherLink III `3C509` Ethernet adapter for ISA/EISA bus.

It provides the following `ioctl` functions:

`SIOCSIFADDR`	Used by the `ifconfig` utility to pass-on the Internet address of the Ethernet interface.
`SIOCSIFFLAGS`	Used by the `ifconfig` utility to pass-on the flags to set/reset the interface.
`GE_SET_DEV`	Sets the interface in `raw_ethernet` mode. In this mode, the applications can directly communicate to the interface using `raw_ethernet` sockets.
`GE_SET_ADDR`	Changes the Ethernet address of the interface.
`GE_SET_PROTO`	Facilitates the setting up of the protocol range that the `raw_ethernet` socket needs to receive.
`GE_SET_MCAST`	Allows the user to set up a list of “multicast addresses” that the interface can receive as destination node using a `raw_ethernet` socket.
`GE_PROM`	When programming the interface to receive all packets, this `ioctl` utility determines whether or not it is meant for this node.
`GE_DEBUG_MODE`	Puts the interface in debug mode. The data transfers through this interface would be displayed in the debug terminal.
`GE_GET_CTRS`	Gets the `raw_ethernet` interface statistics.
`GE_GET_SADDR`	Gets the current Ethernet address of the interface.

`GE_GET_FADDR`	Gets the station address as assigned by the manufacturer of the ethernet interface hardware.
`GE_CLRIFNAME`	Closes the interface for `raw_ethernet` reception for a `raw_ethernet` socket.

The `/sys/devices/if_3c509info.c` file has the following structure for configuring the Ethernet interface:

```
struct if_3c509_info {
    long iobase;   /* io base address ISA: 0x200-0x3E0,
                       EISA: 0x3F0 */
    long vector;   /* 32+IRQ */
    int slot;      /* EISA slot number */
    int xcvr_type; /* TP = 0, AUI = 1, BNC = 3 */
    int bus;           /* bus type, PCMCIA=3, EISA=2, ISA=1 */
};
```

FILES

`/sys/devices/if_3c509info.c`

`/dev/el_isa`

SEE ALSO

Utility Programs -	`ifconfig`
Device Facilities -	`raw_ethernet`
Documents -	See “Raw Ethernet Support” in the LynxOS User’s Guide

NOTES

Unlike normal devices, this device cannot be operated with open/close `ioctl` calls directly. Instead, the device can be operated through sockets either in `AF_INET` domain or in `AF_RAWETH` domain.

NAME

`floppy` – floppy disk device driver

DESCRIPTION

The floppy disk driver controls up to two 3.5 inch floppy drives. The minor device numbers should be 0 for the first drive and 1 for the second.

The `ioctl` system call can be used to find out the capacity of a disk or format a track on the disk. The first argument should be a file descriptor open on a raw or block special file. The second argument can be one of the constants `BLCK_CAPACITY` or `BLCK_FORMAT`, defined in `ioctl.h`. The `BLCK_CAPACITY` command returns the number of 512-byte blocks are available in a long word pointed to by the third argument:

```
int fd;
long blocks;
fd = open("/dev/floppy0", O_RDONLY, 0);
ioctl(fd, BLCKCAPACITY, &blocks);
```

Note that the driver currently always returns 1440 as the number of blocks.

The `BLCK_FORMAT` command will format one track of the floppy. The third `ioctl` argument should point to a structure of type `struct fmtflop`, defined in `ioctl.h`:

```
struct fmtflop {
    char *buffer;       /* pointer to track buffer (8k) */
    int device_num;     /* not used */
    int track;          /* track to be formatted (0-79) */
    int side;           /* 0 or 1 */
    int spt;            /* number of sectors per track */
    int interlv;        /* interleave factor (1-8) */ ;
};
```

The `BLCK_FORMAT` request is used in the utility program `fmtflop` and is not normally necessary in applications programs.

SEE ALSO

Utility Programs - `fmtflop`

System Calls - `ioctl()`

NAME

`hawk` – interface to Ethernet

DESCRIPTION

The `hawk` device driver is a special character device to perform communication over Ethernet/IEEE802.3 data link. This driver supports the VME-based V/Ethernet 3207 LAN controller board. It provides the following `ioctl` functions:

`SIOCSIFADDR`	Used by the `ifconfig` utility to pass-on the internet address of the Ethernet interface.
`SIOCSIFFLAGS`	Used by the `ifconfig` utility to pass-on the flags to set/reset the interface
`GE_SET_DEV`	This `ioctl` call is used to set the interface in `raw_ethernet` mode. In this mode, the applications can directly communicate to the interface using `raw_ethernet` sockets.
`GE_SET_SADDR`	This `ioctl` would be used to change the Ethernet address of the interface.
`GE_SET_PROTO`	This `ioctl` facilitate the setting up of the "protocol range" that the `raw_ethernet` socket needs to receive.
`GE_SET_MCAST`	This `ioctl` call allows the user to setup a list of "multicast addresses" that the interface can receive as destination node, using a `raw_ethernet` socket.
`GE_PROM`	This `ioctl` would realize in programming the interface to receive all packets, whether or not it is meant for this node.
`GE_DEBUG_MODE`	This `ioctl` call would put the interface in debug mode. The data transfers through this interface would be displayed in the debug terminal.
`GE_GET_CTRS`	This `ioctl` call would get the `raw_ethernet` interface statistics.
`GE_GET_SADDR`	This `ioctl` call gets the current Ethernet address of the interface.

GE_GET_FADDR	This ioctl would get the station address as assigned by the manufacturer of the Ethernet interface hardware
GE_CLRIFNAME	This ioctl call would close the interface for raw_ethernet reception for a raw_ethernet socket.

The /sys/devices/if_hawkinfo.c file will have the following structure to configure the Ethernet interface.

```
struct hwkinfo {
    long port;    /* hwk registers in short i/o space */
    long base;    /* Base VMEbus address of onboard mem */
    long vector;  /* Lance Interrupt Vector */
    int intpri;   /* Interrupt level of Lance */;
};
```

FILES

/sys/devices/if_hwkinfo.c

/dev/hawk

NOTE

Unlike normal devices, this device cannot be operated with open, close, ioctl calls directly. Instead, the device can be operated through sockets either in AF_INET domain or in AF_RAWETH domain.

SEE ALSO

Utility Programs -	ifconfig
Device Facilities -	raw_ethernet
Documents -	See “Raw Ethernet Support” in the LynxOS User’s Guide

NAME

`hbtcpip0` – TCP/IP protocol stack driver

DESCRIPTION

The `hbtcpip0` driver is a special character device, that implements the TCP/IP protocol processing. This driver facilitates TCP, UDP, RAW, IP, and RAWETH protocol processing. It also supports `socket` interface for user access.

The `/sys/device/hbtcpip_info.c` file will have the following structure to configure the memory usage by the TCP/IP driver.

```
struct hbtcpip_info {
    unsigned long n_mbuf;
    unsigned long n_clstr;
}
```

The `n_mbuf` field indicates the number of `MBUF`s configured for TCP/IP usage and the field `n_clstr` indicates the `CLUSTER` usage. Each `MBUF` is 256 bytes long (240 bytes for data storage and 16 bytes for the header) and each `CLUSTER` 1024 bytes long. The memory allocation happens during the `HBTCPIP` driver installation. Also, these parameters set the upper limit for TCP/IP memory usage. The default values in:

```
/sys/devices/hbtcpip_info.c
```

indicates 512 `MBUF`s (occupy 512*256 = 128KB) and 128 `CLUSTER`s (128KB) for TCP/IP usage.

FILES

`/sys/devices/hbtcpip_info.c`

`/dev/hbtcpip0`

SEE ALSO

Device Facilities - `wd, oblan`

Documents - See "Raw Ethernet Support" in the LynxOS User's Guide

NOTE

Unlike normal devices, this device cannot be directly operated with the `open`, `close`, or `ioctl` calls. The device can be operated through sockets either in the `AF_INET` domain or the `AF_RAWETH` domain.

NAME

`hd` – ATA-2 (IDE/EIDE) and ATAPI driver (x86)

DESCRIPTION

The `hd` driver can control up to two AT Attachment Interface (ATA-2) devices per channel and supports ATA Packet Interface (ATAPI) CD-ROM devices. The `hd` driver supports the primary and secondary channels. The secondary channel device names start with `hds`. The driver automatically detects whether a device is an ATAPI device or an ATA-2 device at open. The `hd` driver also supports Logical Block Address mode and automatically translates disks according to the Phoenix translation method.

The `hd` driver only supports ATA-2 devices that support the `identify device` command. All EIDE drives must have this command to be compliant. This command allows the driver to determine the geometry of the disk without having to resort to the use of the BIOS type 47 drive tables. Unfortunately, the BIOS uses the numbers in the drive tables so the numbers in the drive table must match physical numbers on the disk, or the translated ones.

The `hd` driver supports two ioctls:

`BLCK_CAPACITY` Returns number of 512 byte blocks on device.

```
int blocks;
ioctl(fd,BLCK_CAPACITY,&blocks);
```

`BLCK_INFO` Returns fields defined in geometry.

```
struct geometry diskinfo;
/* Defined in /sys/dheaders */
ioctl(fd,BLCK_INFO,&diskinfo);
struct geometry {
    short cylinders;
    unsigned char heads;
    unsigned char sectors;
    short write_precomp_cylinder;
    short landing_zone;
};
```

EXTENDED DESCRIPTION

Translation

Drive geometry interpretation really isn't an issue unless the disk is also a boot disk, or when the disk is being shared with another operating system. If two pieces of software are not using the same drive geometry, you may see problems ranging from failure to boot to data corruption. To avoid such problems, don't use multiple operating systems on the same disk.

Booting

Failure to boot can be a more complex problem. This section describes the boot process, highlighting potential problem areas.

When a computer boots, it loads in Block 0 of the bootable disk and executes a program located there. This very small program must use INT 0x13 BIOS calls to read in other blocks from the disk. Block 0 also contains the partition table, allowing the small program to scan for the active partition, partition starting head, starting sector, and starting cylinder. It passes these values to the INT 0x13 BIOS call and loads one 512 byte block into memory.

This code is called the secondary loader. This secondary loader contains a list of blocks to load into memory. In LynxOS, this list of blocks is a program called `preboot`. The list of blocks are loaded into memory using the INT 0x13 BIOS call, limited to 512 bytes (not enough memory for a driver). Once all the blocks are loaded into memory they are executed. This is how `preboot` starts to run. At this point, `preboot` contains the drivers that Lynx supports and handles the rest of the boot process.

It is important to note that until `preboot` is loaded, the system BIOS performs all the disk I/O. A disk must conform to BIOS in order to be bootable.

EXAMPLES

Examples of IDE devices:

`hd0`	Primary IDE controller, master drive
`hd1`	Primary IDE controller, slave drive

`hds0`	Secondary IDE controller, master drive
`hds1`	Secondary IDE controller, slave drive

Mounting a CD-ROM in an ATAPI secondary master device.

```
mount /dev/hds0 /mnt -r
```

SEE ALSO

Utilities - `mkpart, scandisk`

System Calls - `ioctl()`

NOTES

LynxOS works best when the ATAPI device is a secondary master (`hds0`).

If the BIOS translation method does not match the Phoenix proposed method, you will not be able to boot the disk.

NAME

`hybrid_ethernet` – STREAMS and BSD network interface

SYNOPSIS

```
#include <dlpi.h>

#include <dlpiuser.h>
```

DESCRIPTION

The `hybrid_ethernet` interface is an Ethernet driver facility which supports both the STREAMS-based DLPI interface and BSD Network interface for TCP/IP protocol support. This facility is required in LynxOS to support the TCP/IP protocols which require the BSD Network interface and other networking protocols which are STREAMS-based and require a DLPI interface. To install this `hybrid_ethernet` interface facility it is required to install both the TCP/IP and STREAMS products into the kernel.

The `hybrid_ethernet` interface facility consists of three components:

1. A STREAMS-based Ethernet driver providing DLPI Interface
2. A Pseudo BSD Network Interface driver
3. A utility called `dlbsd`

STREAMS-based Ethernet Driver

The STREAMS-based Ethernet driver controls the Ethernet interface hardware and supports multiple streams with DLPI interface. For details of the DLPI interface refer to *Data Link Provider Interface Specification*, Revision 2.0.0 document from UNIX International. The DLPI V2 specification enables a data link service user to access and use any of a variety of conforming data link service providers without special knowledge of the provider's protocol.

The interface specifies access to the data link service provider using the `M_PROTO` and `M_PCPROTO` type of STREAMS messages. It defines the syntax and semantics of the primitives exchanged between the data link user and the data link provider to bind a data link-level address to the stream, get implementation-specific information from the data-link provider, send/receive link-level user data, enable/disable multi-cast group and promiscuous mode reception, get/set the physical addresses associated with link-level station and others. The following is the implementation specific information for the Ethernet driver, specified in the Appendix F of the DLPI Specification document:

- Interface Style

  Implements “style 1” type of interface. It does not require the use of “attach” and “detach” primitives. The stream is bound to a specific PPA based on the device node name opened.

- Service Mode

  Supports connection-less service mode.

- DLSAP address space

  - The DLSAP address length is 8-bytes. The DLSAP address format consists of the 6-byte Ethernet address followed by the 2-byte SAP component. The SAP field is interpreted as an Ethernet packet “type” field. Therefore the valid values for the SAP field are in the [`0-0xFFFF`] range.
  - The same SAP value may be bound to multiple streams. In that case, a received packet may be routed to multiple streams.
  - In addition to the Ethernet V2 service, an 802.3 mode is also supported by the driver. The SAP values in the range [`1-1500`] are treated as equivalent and represent an indication from the user to use the 802.3 mode. If the user wants to send a frame in 802.3 mode the SAP component in the destination DLSAP address specified with the `DL_UNITDATA_REQ` primitive should be equal to the length of the data part of the frame. All frames received from the media having a “type” field in this range are assumed to be 802.3 frames and routed up all open streams which are bound to any SAP value within this range.

- Subsequent DLSAP Addresses

  - The `DL_SUBS_BIND_REQ`, `DL_SUBS_BIND_ACK` and `DL_SUBS_UNBIND_REQ` primitives are supported. The “peer” addressing is supported. So, it is possible to bind multiple SAP addresses to a stream. The syntax and semantics of the subsequent SAP address is the same as that of the SAP address bound with `DL_BIND_REQ` primitive. The SAP should be specified as an `ulong` type field in the `DL_SUBS_BIND_REQ` primitive, and should be within the range [0 - 0xFFFF].

- The maximum number of subsequent addresses that can be bound to a stream is an user-configurable parameter.

- PPA Access and Control
  - Each PPA (Physical Point of Attachment) is configured as a clone-able STREAMS device. The maximum number of streams that may be open for a device is an user-configurable parameter.
  - The `DL_ATTACH` and `DL_DETACH` primitives are not supported. When the STREAMS device is opened, the stream is automatically associated with the PPA for that device.
  - The PPA is automatically initialized when the first stream to that device is opened.
  - The PPA is automatically de-initialized when the last stream to that device is closed.
- Quality of Service

  No support for this service is provided.
- `DL_INFO_ACK` Values

The values returned by the driver in the DL_INFO_ACK primitive in response to the DL_INFO_REQ from the user are as follows:

- The maximum SDU is 1500
- The minimum SDU is 1
- The DLSAP address length is 8
- The MAC type is DL_CSMACD
- The SAP length value is -2 indicating that the physical address component is followed by 2-byte SAP component within the DLSAP address.
- The service mode is DL_CLDLS.
- The QOS fields are 0 indicating that no optional quality of service is supported.
- The provider style is DL_STYLE1
- The version is DL_VERSION2. The broadcast address value is 0xFFFFFFFF

Operation of the STREAMS Driver

When the device is opened, a stream is created and put in DL_UNBOUND state. If this is the first stream opened on that device, the device hardware is initialized. Next, the DL_BIND_REQ primitive is used to bind a DLSAP to the stream. The SAP value bound will determine the “type” of Ethernet packets which will be received on this stream. When the bind operation is done successfully, the stream is placed in DL_IDLE state and a DL_BIND_ACK primitive is sent back to the user. Only when the stream is in DL_IDLE state, it is possible to send/receive packets through that stream.

The user can bind additional SAP addresses to the stream using the DL_SUBS_BIND_REQ primitive in the DL_IDLE state.

Once in the DL_IDLE state, the user may transmit packets on the Ethernet by using the DL_UNITDATA_REQ primitive. The driver will route the received Ethernet packets up all the open streams which are in the DL_IDLE state and bound to SAPs matching the “type” field of the received packet. Received Ethernet packets are duplicated and routed up multiple streams if necessary. The DLSAP addresses contained within the DL_UNITDATA_REQ and DL_UNITDATA_IND messages consists of both the SAP (type) and physical (Ethernet) components.

In addition to the above basic primitives, the STREAMS driver also supports the following optional primitives.

- The DL_ENABMULTI_REQ / DL_DISABMULTI_REQ primitive enables/disables reception of individual multi-cast group addresses. A set of multicast addresses may be associated with a stream by repeatedly using the DL_ENABMULTI_REQ primitive. The maximum number of multi-cast addresses that may be bound to a stream is an user-configurable parameter for the driver. These primitives can be issued in any state after the stream is opened.
- The DL_PROMISCON_REQ / DL_PROMISCOFF_REQ primitive with the DL_PROMISC_PHYS flag set in the dl_level field enables/disables reception of all packets on the Ethernet medium including the packets generated by the local host. When used with the DL_PROMISC_SAP flag set, this enables/disables reception of all SAP values. When used with the DL_PROMISC_MULTI flag set, this enables/disables reception of all multi-cast group addresses.
- The DL_PHYS_ADDR_REQ primitive is used to get the 6-byte Ethernet address currently associated with the PPA.
- The DL_SET_PHYS_ADDR_REQ primitive can be used to change the physical Ethernet address associated with the PPA for this stream. This operation is allowed only when there is no STREAM which is in DL_IDLE state and the Pseudo BSD Network Interface is not attached to the STREAMS device. Once changed, the physical address will remain so until it is changed again or the system is re-booted.
- The DL_GET_STATISTICS_REQ primitive can be used to get the Ethernet packet transmission/reception statistics. The information that is returned in the DL_GET_STATISTICS_ACK primitive should be interpreted according to the lan_statistics structure defined in the dlpiuser.h header file.

Support For Pseudo BSD Network Interface Driver

The STREAMS driver provides the following `ioctl` commands to control the association with the pseudo BSD network Interface driver which handles the packet transmission/reception of the packets for TCP/IP protocol code:

```
DLPIBSD_ATTACH
DLPIBSD_DETACH
DLPIBSD_STATUS
```

These commands are defined in the header file `<dlpiuser.h>`. These `ioctl` commands can be sent down to the driver using an `I_STR` type of `ioctl` and a pointer to a `strioctl` structure.

The `DLPIBSD_ATTACH` command is used to attach a specified Pseudo BSD Network Interface with the PPA of the DLPI stream. The `ic_dp` field in the `strioctl` structure should point to a character string containing the name of the Pseudo BSD Network Interface and the `ic_len` field should contain the length of this name. After the successful return from this ioctl operation, the stream may be closed without breaking the “attachment”.

The `DLPIBSD_DETACH` command is used to detach the PPA of the stream from a previously attached Pseudo BSD Network Interface. It does not require any arguments in the `strioctl` structure.

The `DLPIBSD_STATUS` command is used to check whether the PPA is attached to any Pseudo BSD Network Interface or not. The `ic_dp` field in the `strioctl` structure should point to a `ulong` size data structure and the `ic_len` field should contain that length. The driver returns 0 in this data structure if the PPA is not attached to any Pseudo BSD Network Interface, and a non-zero value if it is.

The above `ioctl` commands are used by the utility `dlbsd` which can be used for attach, detach or status-check operations described above.

Pseudo BSD Network Interface Driver

A Pseudo BSD Network Interface driver handles the packet transmission and reception for the BSD TCP/IP protocol layers. Each such interface has an assigned DLPI STREAMS device to use. This device is normally not attached to the DLPI device. Using the `DLPIBSD_ATTACH` `ioctl` command on a stream a Pseudo BSD Network Interface can be attached to the PPA of the assigned STREAMS device. Once a Pseudo BSD Network Interface is attached to a STREAMS device, the `ifconfig` utility can be used to configure the Pseudo BSD Network Interface for TCP/IP networking.

SEE ALSO

Utility Programs - `dlbsd, ifconfig`

Device Facilities - `dlpilan, bdlan, dlpi_21040`

NAME

`icmp` – Internet Control Message Protocol

SYNOPSIS

In the socket environment:

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

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

In the TLI environment:

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

t = t_open("/dev/ip", O_RDWR, info);
```

DESCRIPTION

ICMP is the error and control message protocol used by IP and the Internet protocol family. It may be accessed through a `raw socket` for network monitoring and diagnostic functions. The **proto** parameter to the socket call to create an `ICMP` socket is obtained from `getprotobyname(3)`. `ICMP` sockets are connectionless, and are normally used with the `sendto` and `recvfrom` calls, though the `connect(2)` call may also be used to fix the destination for future packets (in which case the `read(2)` or `recv(2)` and `write(2)` or `send(2)` system calls may be used). In the TLI environment it can be accessed by opening the STREAMS device `/dev/ip`. The stream must be bound to `IPPROTO_ICMP`. The address is a single byte in length. The `t_sndudata(3)` and `t_rcvudata(3)` calls then may be used.

Outgoing packets automatically have an `IP` header prepended to them (based on the destination address). Incoming packets are received with the `IP` header and options intact.

DIAGNOSTICS

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

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

SEE ALSO

System Calls -	`send(), recv()`
Library Functions -	`t_open(), t_bind(), t_sndudata(), t_rcvudata()`
Device Facilities -	`intro, inet, ip`

NAME

`inet` – internet protocol family

SYNOPSIS

`#include <sys/types.h>`

`#include <netinet/in.h>`

DESCRIPTION

The Internet protocol family is a collection of protocols layered atop the Internet Protocol (IP) transport layer, and utilizing the Internet address format. In the socket environment the Internet family provides protocol support for the `SOCK_STREAM`, `SOCK_DGRAM`, and `SOCK_RAW` socket types; the `SOCK_RAW` interface provides access to the `IP` protocol. In the TLI environment, it provides support for `T_COTS` and `T_CLTS` type of transport service.

ADDRESSING

Internet addresses are four byte quantities, stored in network standard format (on the 386, these are word and byte reversed). The `include` file `<netinet/in.h>` defines this address as a discriminated union.

Sockets and the TLI transport endpoints 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 local address `INADDR_ANY` to effect wildcard matching on incoming messages. The address in a `connect(2)` or `sendto(2)` call may be given as `INADDR_ANY` to mean “this host.” The distinguished address `INADDR_BROADCAST` is allowed as a shorthand for the broadcast address on the primary network, if the first network configured supports broadcast.

PROTOCOLS

The Internet protocol family is comprised of 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` and `T_COTS` abstraction while `UDP` is used to support the `SOCK_DGRAM` and `T_CLTS` abstraction. A raw interface to `IP` is available by creating an Internet socket of type `SOCK_RAW` or directly opening the `/dev/ip` device in the TLI environment. The `ICMP` message protocol is accessible from a raw socket or from opening the STREAMS device `/dev/ip`.

The 32-bit Internet address contains both network and host parts. It is frequency-encoded; the most-significant bit is clear in Class A addresses, in which the high-order 8 bits are the network number. Class B addresses use the high-order 16 bits as the network field, and Class C addresses have a 24-bit network part. Sites with a cluster of local networks and a connection to the DARPA Internet may chose to use a single network number for the cluster; this is done by using subnet addressing.

The local (host) portion of the address is further subdivided into subnet and host parts. Within a subnet, each subnet appears to be an individual network; externally, the entire cluster appears to be a single, uniform network requiring only a single routing entry. Subnet addressing is enabled and examined by the following `ioctl(2)` commands on a datagram socket in the Internet domain; they have the same form as the `SIOCIFADDR` command (see `intro(4)`).

DIAGNOSTICS

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

`SIOCGIFNETMASK` Get interface network mask.

CAVEAT

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

SEE ALSO

System Calls - `ioctl(), socket()`

Device Facilities - `intro, tcp, udp, ip, icmp`

NAME

`ioctl –` SCSI ioctl command

DESCRIPTION

The `ioctl` system call is used to directly access SCSI devices. This manual page details the `ioctl` interface to SCSI devices. For the synopsis of valid parameters to `ioctl`, see the `ioctl(2)` manual page.

The first parameter to `ioctl` is an open file descriptor to either the raw or block SCSI device. The second parameter can be one of the constants `BLCK_CAPACITY` or `SCSI_ANYTHING` (defined in `ioctl.h`). The request `BLCK_CAPACITY` writes the number of blocks available on the device into the long integer pointed to by the third argument to `ioctl`. These blocks are not necessarily 512 bytes in size. Note that `BLCK_CAPACITY` does not return the size of the blocks.

```
int fd;
long blocks;
fd = open("/dev/sd0", O_RDONLY, 0);
ioctl(fd, BLCK_CAPACITY, &blocks);
```

Though not needed for normal operation, it is possible to give virtually any command to a SCSI device using the `SCSI_ANYTHING` command. For this request, the third `ioctl` parameter should be of type `struct scblock`, defined in `scsi.h`:

```
struct scblock {
    unsigned char command[23];
    unsigned char target_status;
    short cmdlen;
    int size;
    short direction;
    char sense_data_valid;
    unsigned char sense_data[128];
    unsigned char data[256];
};
```

Description of fields of `scblock`:

`command` Contains the command to perform. Opcodes for commands are contained in `scsi.h`.

`target_status` Not set by the user.

`cmdlen` Only set by the user when a vendor specific SCSI command is sent (Group 6 and 7). For all other groups, this will be set by the drivers.

`size` Number of bytes of data to transfer. This size should not be larger then the size of the field data. However, one can redefine the `scblock` structure by increasing `data` to a maximum of 4096. The driver will not allow more than 4096 bytes to be transferred to the `scblock`.

`direction` Tells the data flow direction.

- `TO_SCSI` Data in the data field will be transferred to the host adaptor as indicated by the command.
- `FROM_SCSI` Data will be transferred from the host adaptor to the data field as indicated by the command.

`data` Contains information to be sent or information that was received by running the command. Sometimes the data to be sent or received will be more then 256 bytes. This structure can be declared in the user program with a larger data size.

`sense_data_valid` Set to 1 if `sense_data` contains valid information. This is only supported with `scsimgr`.

`sense_data` Contains information specific to the command given. Consult a SCSI manual to know how to interpret the return values. This is only supported with `scsimgr`.

EXAMPLE

This example shows how `SCSI_ANYTHING` can be used to implement a routine that is already provided by default in the SCSI drivers.

```
int get_blocks(int *num_blocks, int *block_size)
{
    struct scblock s;
    int status;

    bzero((char *) &s, sizeof(struct scblock));
    s.command[0] = READ_CAPACITY;
    s.direction = FROM_SCSI; /* get data from host adapt.*/
    s.size = 8;     /* transfer 8 bytesfrom host adapter */
    status = ioctl(fd, SCSI_ANYTHING, &s);
    if (!status) {      /* ioctl returns 0 for ok status */
        *num_blocks = s.data[0] << 24 | s.data[1] << 16 |
            s.data[2] << 8 | s.data[3];
        *block_size = s.data[4] << 24 | s.data[5] << 16 |
            s.data[6] << 8 | s.data[7];
    }
    return status;
}
```

The `SCSI_ANYTHING` request is generally used for utility software which must manipulate devices directly: for example, to format disks. It is only available to processes with effective user ID 0 (root).

SEE ALSO

Utility Programs - `fmtscsi`, `scandisk`, `mkpart`

System Calls - `ioctl()`

Device Facilities - `scsimgr`

NAME

`ip, IP –` Internet Protocol

SYNOPSIS

In the socket environment:

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

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

In the TLI environment:

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

t = t_open("dev/ip", O_RDWR, info);
```

DESCRIPTION

IP is the transport layer protocol used by the Internet protocol family. Options may be set at the `IP` level when using higher-level protocols that are based on `IP` (such as `TCP` and `UDP`). It may also be accessed through a raw socket when developing new protocols, or special-purpose applications.

A single generic option is supported at the `IP` level, `IP_OPTIONS`, that may be used to provide `IP` options to be transmitted in the `IP` header of each outgoing packet. Options are set with `setsockopt(2)` and examined with `getsockopt(2)`. The format of `IP` options to be sent is that specified by the `IP` protocol specification, with one exception: the list of addresses for Source Route options must include the first-hop gateway at the beginning of the list of gateways. The first-hop gateway address will be extracted from the option list and the size adjusted accordingly before use. `IP` options may be used with any socket type in the Internet family.

Raw `IP` sockets are connectionless, and are normally used with the `sendto` and `recvfrom` calls, though the `connect(2)` call may also be used to fix the destination for future packets (in which case the `read(2)` or `recv(2)` and `write(2)` or `send(2)` system calls may be used).

If **proto** is 0, the default protocol IPPROTO_RAW is used for outgoing packets, and only incoming packets destined for that protocol are received. If **proto** is non-zero, that protocol number will be used on outgoing packets and to filter incoming packets.

Outgoing packets automatically have an IP header prepended to them (based on the destination address and the protocol number the socket is created with). Incoming packets are received with IP header and options intact.

In the TLI environment, the device /dev/ip is used to implement the Internet protocols and the ICMP protocols. It is not a STREAMS multiplexing driver. The /dev/ip device is a clonable STREAMS device. The number of minor devices for this driver can be specified by the SMD_NIP parameter in the /sys/devices/stsocket_info.c file. It implements a connectionless protocol and uses the same transport service interface as the TLI T_CLTS interface.

DIAGNOSTICS

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

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

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

EINVAL	An unknown socket option name was given.

`EINVAL`	The `IP` option field was improperly formed; an option field was shorter than the minimum value or longer than the option buffer provided.

SEE ALSO

System Calls -	`getsockopt()`, `send()`, `recv()`,
Library Fuctions -	`t_open()`, `t_bind()`, `t_sndudata()`, `t_rcvudata()`
Device Facilities -	`intro`, `icmp`, `inet`

NAME

`iso9660` – ISO9660 or ISO9660 + RRIP file system

SYNOPSIS

DESCRIPTION

This is the ISO9660 file system with support for the Rock Ridge Interchange Protocol (RRIP) extensions. The `iso9660` driver automatically detects whether the RRIP extensions are present, and if they are uses them in interpreting information. If RRIP extensions are available they will always be used. In this document, ISO9660 will be used to represent an ISO9660 or an ISO9660 + RRIP file system.

This file system is a dynamic driver or a static driver. It is linked in by default in `CONFIG.TBL`. However if it is not linked initially it can be dynamically linked after the kernel is running. This file system has all the usual entry points as a dynamic driver must have, however the only entry point to do anything is `install()`. All the others return `SYSERR`. The file system is a dynamic driver only for purposes of loading extra capabilities into the kernel after boot up. It does not need any nodes in `/dev`.

To mount an ISO9660 file system use the mount command. ISO9660 or RRIP interpetation is done automatically by the driver. There is no need to tell mount what kind of CD you are mounting. No special flags or options are necesary. If the CD-ROM drive is device `/dev/sd2940.1` do the following to mount the ISO9660 volume:

```
mount /dev/sd2940.1 /mnt -r
```

NOTE: The ISO9660 or RRIP CD's do not have partitions so mount the block device without using any partitions.

A disk partition may contain an ISO9660 file system. An example of mounting a partition that contains an ISO9660 file system:

```
mount /dev/sd2940.0a /mnt -r
```

Note that the only difference in the usual mount of a parititon and the mounting of an ISO9660 partition is the extra **-r** (readonly) flag.

BUGS

Uninstalling the `iso9660` driver could cause the kernel to hang or crash.

NAME

`lock` – NFS File Locking driver

DESCRIPTION

The `lock` device driver facilitates the advisory file locking over NFS. Advisory file locking will work fine only with co-operating processes. The file locking will be supported using `fcntl()` and `lockf()` system calls. This device driver supports monitored class of locking, where all the locks will be monitored from both Client and Server side.

The `/sys/devices/lock_info.c` file controls the number of NFS file locking client requests, that will be enqueued in the driver.

```
struct lock_info {
    int nports; /* max # of requests that can be enqueued */
} ;
```

Each entry that corresponds to the request queue takes 3068 bytes. The default value for this field is set as 10. (total memory allocated 10*3068 bytes).

FILES

`/sys/devices/lockinfo.c`

`/dev/lock`

SEE ALSO

Utility programs - `rpc.lockd.svc`, `rpc.lockd.clnt`, `rpc.statd`

NAME

`lynxfs` – LynxOS Fast Lynx File System

DESCRIPTION

This is the LynxOS Fast Lynx File System driver. If this driver is installed, Lynx FLFS file systems can be read.

This file system is a dynamic driver or a static driver. It is linked in by default in `CONFIG.TBL`. However if it is not linked initially it can be dynamically linked after the kernel is running. This file system has all the usual entry points as a dynamic driver must have, however the only entry point to do anything is `install()`. All the others return `SYSERR`. The file system is a dynamic driver only for purposes of loading extra capabilities into the kernel after boot up. It does not need any nodes in `/dev`.

BUGS

Uninstalling the `lynxfs` driver could cause the kernel to hang or crash.

NAME

mem – main memory

SYNOPSIS

#include <mem.h>

DESCRIPTION

mem is a special file that is an image of main memory. It can be read or written like any other file. Addresses in mem are treated as physical memory addresses.

EXAMPLES

A programming example:

```
unsigned long freepages;
unsigned long offset;
fd = open ( "/dev/mem", O_RDONLY, 0 );
/* seek in memory to the address of the variable
that contains the number of free pages of memory
in the system.*/
offset = info(I_FREEPAGES);
if ( lseek ( fd, offset, 0) != offset ) {
    perror ("lseek: ");
    close (fd);
    exit (-1);
}
if ( read (fd, &freepages, sizeof(freepages)) != /
    sizeof(freepages)) {
    perror ("read: ");
    close (fd);
    exit (-2);
}
printf("Number of freepages = %d\n", freepages);
close (fd);
```

FILES

/dev/mem

NAME

`ncr_drivers` – 810/825 & 710/720 NCR device drivers

DESCRIPTION

The `ncr_drivers` are a collection of drivers that, when put together, provide the ability to use the various NCR chips supported by LynxOS. LynxOS currently supports the NCR 810/825 and 710/720 chipsets.

The general layout of `ncr_drivers` includes a low-level driver that knows how to communicate with the hardware together with some high-level drivers that are specifically designed to communicate with a particular SCSI-2 device type. The table below lists these SCSI-2 devices and indicates whether or not they are supported by the `ncr_drivers`. The column "Number" is the SCSI-2 specification number for each device type..

SCSI-2 Device Types	Number	Supported
Direct Access Device	0	yes
Sequential Access Device	1	yes
Printer Device	2	no
Processor Device	3	no
Write Once Device	4	no
CD-ROM Device	5	yes
Scanner Device	6	no
Optical Memory Device	7	no
Medium Changer Device	8	no
Communications Device	9	no

Currently, there are two high-level drivers: `hdscsi` and `mtscsi`. `hdscsi` is a "direct access" high-level driver. `mtscsi` is a "sequential access" high-level driver. `hdscsi` also supports CD-ROM devices, since the functionality of this device for reading is identical to other direct access devices. to `hdscsi`. To access CD-ROM media, use the `hdscsi` nodes.

Each driver has its own major number. Thus, three drivers are needed to get direct access and tape support with the `ncr_drivers`. There are no nodes for the low-level SCSI driver. Only the high-level drivers communicate with the low-level driver. This allows you to add a custom SCSI driver by copying an existing high-level driver and modifying it for the new driver. Unfortunately, there are a lot of redundant algorithms in the high-level drivers.

Direct Access Devices

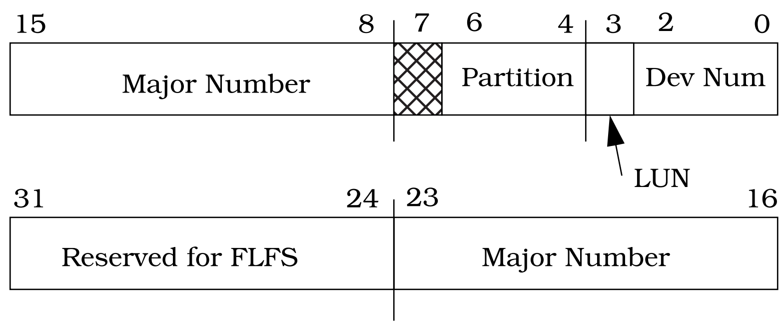
"Direct Access" devices include hard disks and other devices that report as a direct access device from an inquiry command. In addition, CD-ROM devices have many of the same properties as direct access devices and will be covered in this section.

The following figure describes the device node structure of direct access and CD-ROM device nodes. The field "Dev Num" is the ID of that device on the SCSI bus. "Partition" is a number that corresponds to the partition. The field "Major Number" is the number of the major device as displayed using the `devices` command.

NOTE: There is room for up to 8 partitions using `hdscsi`.

`hdscsi` supports devices with physical block sizes other then 512 bytes. When the device is accessed in raw mode, the request size must be a multiple of the physical block size.

Device Node Layout for Direct Access and CD-ROM Devices



Partition	Val
No Partition	0
Partition A	1
Partition B	2
Partition C	3
Partition D	4

CD-ROM devices consist of various CD-ROM readers on the market. These devices are always treated as read only. LynxOS supports raw and block access of CD-ROMs. However, although block support allows access to a LynxOS file system, ISO-9660 is not yet supported by LynxOS. This is due to a current limitation in the file system, not `hdscsi`.

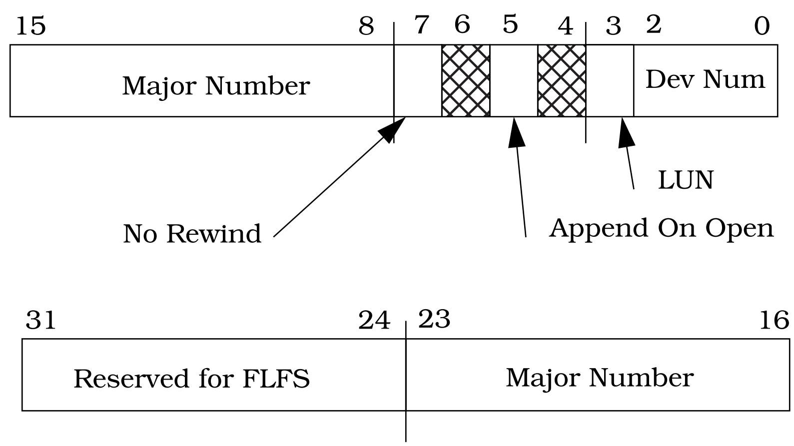
Sequential Access Devices

“Sequential Access” devices consist of various tape devices: DAT, Helical Scan, 150 megabyte QIC tape, etc. Variable length requests are not supported with `hdscsi`; only block requests are supported. Block mode requires that data be read/written in multiples of 512 bytes.

Below is a table that describes the device node layout for sequential access device nodes. The field “Dev Num” represents the ID of the device on the SCSI bus. LUN is the Logical Unit Number and is almost always 0. If the

"Append on Open" bit is set, the media will seek past the first file mark before reading or writing. When the "NoRewind" bit is set, the media will not rewind on a close.

Device Node Layout for Sequential Access Devices



NAME

`nfs_cache` – NFS client cache driver

DESCRIPTION

The `nfs_cache` device driver is used to cache the NFS client request responses received from the NFS server. This cache helps in controlling the amount of NFS traffic over the wire, and is configurable on a per-NFS-mount basis. The `nfs_cache` driver caches the directory entries, file attributes and file data information depending upon the NFS client accesses done on those files. Any client calls over a cached NFS mount point, will be cross checked with the cache contents for a 'match' and will be routed to the Server only if the information isn't available in the cache. Once this driver is configured in the kernel, the caching can be turned ON/OFF specifying the associated parameters at each NFS mount time. The `sys/devices/nfs_cache_info.c` file helps in tuning the NFS cache memory usage by choosing the parameters in the structure:

```
struct nfs_cache_info {
    int num_attrcache;   /* num attribute cache blocks */
    int num_dnlchdrs;   /* num directory name lookup
                           cache headers */
    int size_dnlcblk;   /* size of each dnlc cache block */
    int num_datacache;  /* num of data cache headers */
    int size_datacache; /* amount of memory for data
                           cache*/
    int hash_tblsize;   /* hash table size for data/dnlc
                           blocks */



/* default mount-time options */
    int blksize;        /* caching block size */
    int dis_cache;      /* disable all caching */
    int dis_datacache;  /* disable file data caching */
    int acregmin;        /* min time in secs for reg files */
    int acregmax;        /* max time in secs for reg files */
    int acdirmin;        /* min time in secs for dir files */
    int acdirmax;        /* max time in secs for dir files */
};
```

The various fields in this structure are explained below:

`num_attrcache` This parameter controls the total number of simultaneous file/directory entries that will be allowed to be cached in this client. Each entry for this parameter consumes 180 bytes. Default value for this field is 512 (total memory consumption 512*180 bytes).

`num_dnlchdrs` This parameter controls the maximum number of directory name lookup buffers that are cached in the client. Each entry corresponding to this field consumes 32 bytes. The default value for this field is 128 (total memory consumption 32*128 bytes). Depending up on the size of the directory files, more than one entry may be used for each directory.

`size_dnlcblk` This parameter controls the amount of buffer allocated for each directory name lookup cache header, to store the filename and it's NFS file handle under that directory. The default value for this field is 512. Each filename entry associated with the directory takes 40+<FileNameLen> bytes.

`num_datacache` This parameter indicates the number of data cache headers that will be allocated for the NFS cache. Effectively, this parameter controls the total number of data blocks (of variable size) that can be cached by the driver. Each entry corresponding to this field consumes a total of 96 bytes. The default value for this field is 128. (total memory consumption 128*96 bytes).

`size_datacache` This parameter controls the total amount of memory in kilobytes that will be allocated for data caching. The default value for this field is 512 (memory usage 512KB).

`hash_tblsize` This parameter controls the amount of "hash table" associated with each entry of the file attribute cache. Thus each file attribute entry will be getting `hash_tblsize` number of pointer's to the associated "directory name lookup entries" (in case of directory files) OR "data cache entries" (for regular files). In case of "directory" files the hashing will be done based on the filenames associated with that directory and in case of

	regular files, the hashing is done based on the block number of the data block. The default value of this field is 32 (32*4 bytes).
`blksize`	This parameter determines the default logical blocksize for data caching. This field will be overridden if read/write block size given during at NFS mount time, for that mount. The default value for this field is 8192.
`dis_cache`	This field, if non zero, will disable system-wide NFS caching. The default value for this field is zero and should remain zero.
`dis_datacache`	This field, if set to non-zero, will disable NFS data caching in the system. This field should remain zero.
`acregmin`	This field specifies the minimum time in seconds that the attribute cache entries will be held with the client before placing in the queue used to update the NFS server. This field applies for the regular files. The default value for this field is 10.
`acregmax`	This field specifies the maximum time in seconds that the file attributes is expected to be sane with the NFS client. Beyond this period, the attribute cache is discarded and is used for refreshing. The default value for this field is 60 seconds.
`acdirmin`	This field specifies the minimum amount of time in seconds, that the directory file attributes are considered to be valid. The attributes will be ready for refresh beyond this duration. The default value for this field is 30.
`acdirmax`	This field specifies the maximum time in seconds that the directory file attributes is expected to be sane. After this time period, the attribute cache entry for the files in this directory, if needed, are refreshed from the NFS Server. The default value for this field is 60.

FILES

`/sys/devices/nfs_cache_info.c`

`/dev/nfs_cache`

SEE ALSO

Utility Programs - `mount`

NAME

`nfssvc` – NFS server I/O driver

DESCRIPTION

The `nfssvc` device driver is a special character device to perform NFS (Network File System) server functions. It provides the following `ioctl` functions:

`NFSSVC_EXPORT` Used by the `exportfs` program to pass information to the kernel about the directories which can be shared over the network.

`NFSSVC_UNEXPORT`
Used by the `exportfs` program to `unexport` directories.

`NFSSVC_GETFH` Used by the `mountd` daemon to get the `NFS file handle` for an exported directory.

`NFSSVC_PROTO` Used by the `nfsd` daemon to send the `NFS protocol` requests received from the client machines to the kernel for execution and to get a response back.

Multiple `nfsd` daemons may use this device in parallel to provide service to multiple `NFS client requests` simultaneously.

The `/sys/devices/nfssvc_info.c` file has the following structure which can be used to configure the parameters for NFS server support:

```
struct nfssvc_info {
    int nexports;      /* num of export table entries  */
    int naccesshosts; /* maximum num of hosts that will
                         have mount access to an
                          exported directory */
    int nroothosts;   /* maximum num of hosts that will
                         have root access to an
                          exported directory */
    int nwritehosts;  /* maximum num of hosts that will
                         have both read and write access
                          to an exported directory */
    int nopens;       /* maximum num of nfsd tasks that
                         can open this device for
                          simultaneous use */
};
```

FILES

/sys/devices/nfssvc_info.c

/dev/nfssvc

SEE ALSO

Utility Programs - exportfs, mountd, nfsd

NAME

oblan – interface to Ethernet (68k)

DESCRIPTION

The oblan device driver is a special character device to perform communication over Ethernet/IEEE802.3 data link. This driver supports the on-board LAN controller of the 68k CPU board.

It provides the following ioctl functions:

SIOCSIFADDR	Used by the ifconfig utility to pass-on the Internet address of the Ethernet interface.
SIOCSIFFLAGS	Used by the ifconfig utility to pass-on the flags to set/reset the interface.
GE_SET_DEV	Sets the interface in raw_ethernet mode. In this mode, the applications can directly communicate to the interface using raw_ethernet sockets.
GE_SET_ADDR	Changes the Ethernet address of the interface.
GE_SET_PROTO	Facilitates the setting up of the protocol range that the raw_ethernet socket needs to receive.
GE_SET_MCAST	Allows the user to set up a list of “multicast addresses” that the interface can receive as destination node using a raw_ethernet socket.
GE_PROM	When programming the interface to receive all packets, this ioctl utility determines whether or not it is meant for this node.
GE_DEBUG_MODE	Puts the interface in debug mode. The data transfers through this interface would be displayed in the debug terminal.
GE_GET_CTRS	Gets the raw_ethernet interface statistics.
GE_GET_SADDR	Gets the current Ethernet address of the interface.
GE_GET_FADDR	Gets the station address as assigned by the manufacturer of the ethernet interface hardware.

GE_CLRIFNAME Closes the interface for raw_ethernet reception for a raw_ethernet socket.

The /sys/devices/if_oblaninfo.c file has the following structure for configuring the Ethernet interface:

```
struct hwkinfo {
    long port;        /* Lance registers */
    long vector;      /* Lance Interrupt Vector */
    int intpri;       /* Interrupt level of Lance */
    long pcc_licr;    /* PCC Interrupt Controller register */
    long pcc_ibvr;    /* Interrupt base vector register */
    long  bb_esap;    /* Ethernet Station address in BB RAM */
};
```

FILES

/sys/devices/if_oblaninfo.c

/dev/oblan

SEE ALSO

Utility Programs - ifconfig

Device Facilities - raw_ethernet

Documents - See “Raw Ethernet Support” in the LynxOS User’s Guide

NOTES

Unlike normal devices, this device cannot be operated with open/close ioctl calls directly. Instead, the device can be operated through sockets either in AF_INET domain or in AF_RAWETH domain.

NAME

`pty` – pseudo terminal driver

DESCRIPTION

The `pty` driver provides support for a device-pair termed a “pseudo terminal.” A pseudo terminal is a pair of character devices, a “master” device and a “slave device”. The slave device provides processes an interface identical to that described in `tty`. However, whereas all other devices which provide the interface described in `tty` have a hardware device of some sort behind them, the slave device has, instead, another process manipulating it through the master half of the pseudo terminal. 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 following `ioctl` calls apply only to pseudo terminals:

`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 pseudo terminal, each subsequent read from the terminal will return data written on the slave part of the pseudo terminal preceded by a zero byte (symbolically defined as `TIOCPKT_DATA`), or a single byte reflecting control status information. In the latter case, the byte is an bitwise-OR of zero or more of the bits:

`TIOCPKT_FLUSHREAD`
: the read queue for the terminal has been flushed.

`TIOCPKT_FLUSHWRITE`
: the write queue for the terminal has been flushed.

`TIOCPKT_STOP`
: output to the terminal is stopped a la `^S`.

`TIOCPKT_START`
: output to the terminal has been restarted.

TIOCPKT_DOSTOP
: t_stopc is ^S and t_startc is ^Q.

TIOCPKT_NOSTOP
: the start and stop characters are not ^S and ^Q.

This mode is used by rlogin and rlogind to implement a remote-echoed, locally ^S/^Q flow-controlled remote login with proper back-flushing of output; it can be used by other similar programs.

FILES

/dev/pty[p-r][0-9, a-f]	Master pseudo terminals
/dev/tty[p-r][0-9, a-f]]	Slave pseudo terminals display

NOTES

It is not possible to send an EOT.

NAME

`raw_ethernet` – Raw Ethernet protocol family

SYNOPSIS

`#include <types.h>`

`#include <bsd/raweth.h>`

DESCRIPTION

The `raw_ethernet` protocol allows the user to communicate from application layer to the ethernet interface layer directly. This facilitates protocol support for the address family `AF_RAWETH` with `SOCK_RAWETH` socket type. The `write` and `read` system calls may be used to send and receive ethernet packets containing the ethernet header and data. The following `ioctl` functions are provided for the `raw_ethernet` type sockets:

`GE_SET_DEV`	This `ioctl` call is used to set the interface in `raw_ethernet` mode. In this mode, the applications can directly communicate to the interface using `SOCK_RAWETH` sockets.
`GE_SET_SADDR`	This `ioctl` would be used to change the “ethernet address” of the interface.
`GE_SET_PROTO`	This `ioctl` facilitate the setting up of the “protocol range” that the `SOCK_RAWETH` socket needs to receive.
`GE_SET_MCAST`	This `ioctl` call allows the user to setup a list of “multicast addresses” that the interface can receive as destination node, using a “`SOCK_RAWETH` socket”.
`GE_PROM`	This `ioctl` would realize in programming the interface to receive all packets, whether or not it is meant for this node.
`GE_DEBUG_MODE`	This `ioctl` call would put the interface in `debug` mode. The data transfers through this interface would be displayed in the debug terminal.
`GE_GET_CTRS`	This `ioctl` call would get the `raw_ethernet` interface statistics.

`GE_GET_SADDR`	This `ioctl` call gets the current ethernet address of the interface.
`GE_GET_FADDR`	This `ioctl` would get the station address as assigned by the manufacturer of the ethernet interface hardware
`GE_CLRIFNAME`	This `ioctl` call would close the interface for `raw_ethernet` reception for a `SOCK_RAWETH` socket.

SEE ALSO

Documents -	See “Raw Ethernet Support” in the LynxOS User’s Guide*x*

NAME

`scsimgr`– common SCSI command algorithm (x86)

DESCRIPTION

The `scsimgr` handles high-level queuing and building of SCSI commands for all low-level SCSI Interface Modules (SIM) in LynxOS. The `scsimgr` removes the need for redundant queueing algorithms within the SIM drivers, thus allowing the SIMs to focus on hardware implementations. The routines in `scsimgr` are called from the SIMs only.

The `scsimgr` uses an inquiry command to determine the type of device being opened. If a device does not support the inquiry command, that device cannot be used with the `scsimgr`. the table "SCSI-2 Devices Supported by scsimgr" lists SCSI-2 devices and indicates whether or not they are supported by the `scsimgr`. The column "Number" is the SCSI-2 specification number for each device type.

Table 1: SCSI-2 Devices Supported by scsimgr

SCSI-2 Device Types	Number	Supported by scsimgr
Direct Access Device	0	yes
Sequential Access Device	1	yes
Printer Device	2	no
Processor Device	3	no
Write Once Device	4	yes
CD-ROM Device	5	yes
Scanner Device	6	no
Optical Memory Device	7	no
Medium Changer Device	8	no
Communications Device	9	no

The `scsimgr` can be stubbed out of the kernel by replacing the following line in the `KERNEL.CFG` file: change `+scsimgr` to `–scsimgr`. You may want to stub out the `scsimgr` if you are building a very small kernel and have no SIM devices configured via `CONFIG.TBL`.

DIRECT ACCESS DEVICES

"Direct Access" devices include hard disks and other devices that report as a direct access device from an inquiry command. In addition, CD-ROM and "Write Once" devices have many of the same properties as direct access devices and will be covered in this section.

The following figure describes the device node structure of direct access, CD-ROM, and write once device nodes. The field "Device Number" is the ID of that device on the SCSI bus. "Partition" is a number that corresponds to the partition. The field "Major Number" is the number of the major device as displayed using the `devices` command.

NOTE: LynxOS provides for up 16 partitions. Only the lower 2 bits of the partition field are used by LynxOS at this time.

Device Node Layout for Direct Access, CD-ROM and Write Once Devices

15 — 8	7 — 4	3 — 0
Major Number	Partition	Device Num

31 — 24	23 — 16
Reserved for FLFS	Reserved (Not Used)

Partition	Val
No Partition	0
Partition A	1
Partition B	2
Partition C	3
Partition D	4

The `scsimgr` supports devices with physical block sizes other then 512 bytes. This only matters when the device is accessed in raw mode because the request size must be a multiple of the physical block size.

CD-ROM Devices

CD-ROM devices consist of various CD-ROM readers on the market. These devices are always treated as read only. LynxOS CD-ROM support consists of raw and block support.

NOTE: Block support allows the LynxOS file system to be accessed via CD-ROM. However, the ISO-9660 standard is not yet supported by LynxOS due to current limitations in the file system.

Write Once Devices

"Write once" devices, such as CD-ROM writers, are recognized by the `scsimgr`. The `scsimgr` supports opening, writing and performing ioctls to these type of devices. This set of features provides enough support that a customized utility can be written to use LynxOS (via the `scsimgr`) to write data onto CD-ROMs.

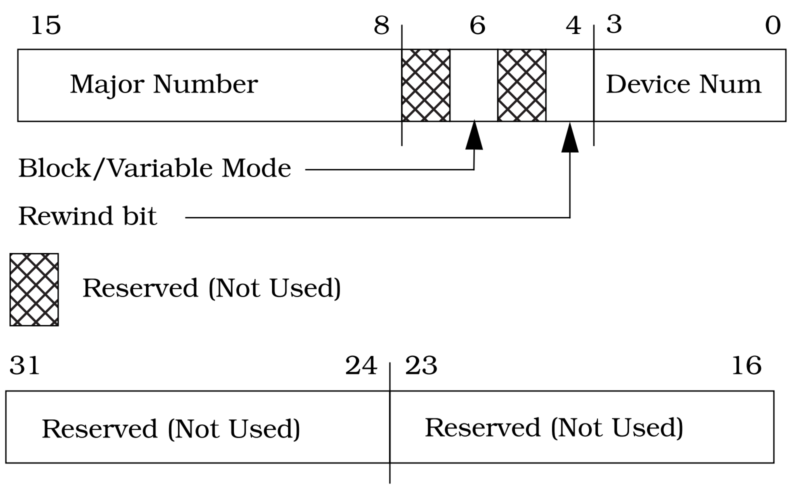
You must specify the starting block to the **`write`** function of the `scsimgr` (usually 0, unless a multi-session CD-ROM is being created). However, write once devices usually don't need this information since the data is written sequentially anyway.

NOTE: Even though write once devices can be opened in block mode for writing, this mode must not be used as the writes will not hit the disk sequentially due to interaction with the disk cache. Write using the raw device only.

SEQUENTIAL ACCESS DEVICES

“Sequential Access” devices consist of various tape devices: DAT, Helical Scan, 150 megabyte QIC tape, etc. Variable and block access of these tape devices are supported. Below is a figure that describes the device node layout for sequential access device nodes. The field “Device Number” represents the ID of the device on the SCSI bus.

Device Node Layout for Sequential Access Devices



When the “Block/Variable Mode” bit is set, the device will be used in block mode. Block mode means that data must be read/written in multiples of 512 bytes. If this bit is not set, then variable mode is selected and data can be processed in units down to the byte. Since not all devices support variable mode, it is important that you refer to your device documentation for more information.

If the “Rewind bit” is set, then the device will be rewound when opened and rewound when closed. If this bit is not set, no rewinding will occur.

SEE ALSO

Device Facilities-	`sim2940, sim2742, sim1742, sim1542, sim1522`
File Format-	`KERNEL.CFG, CONFIG.TBL`

NAME

`sim1522` – Adaptec 1520/1522 SCSI device driver (x86)

DESCRIPTION

The SCSI device driver can control up to 7 devices on the SCSI bus. See the `scsimgr(4)` manual page for details on general devices supported. Target devices can be numbered 0 through 6; the CPU board controller is device 7.

Features of the `sim1522`:

- Uses `scsimgr`.
- Support for disconnection/reconnection.
- Use of programmed I/O instead of interrupt-driven DMA transfers.

Limitations of the `sim1522`:

- Designed for the 6260 chip set, so may or may not work with the 6360.
- Does not support CD-ROM or WORM devices.
- Does not support synchronous devices.

The `info_structure` used by `sim1522` is defined as:

```
struct adapter_info {
    u_char scsi_port;      /* SCSI Bus/Port ID */
    u_char cntrl_id;       /* Controller ID */
    u_char scsi_id;        /* SCSI ID of host adapter   */
    u_short base_address;  /* Adapter Base I/O address  */
    u_short dma_chan;      /* DMA channel used */
    u_short vector;        /* Interrupt Vector */
    u_short level;         /* Interrupt Level */
    u_short performance;
};
```

The elements in this structure are described below:

`scsi_port` Not used.

`cntrl_id`	Not used.
`scsi_id`	ID of host adapter (Should be set to 7).
`base_address`	Port address of host adapter.
`dma_chan`	Not used (driver does not use DMA).
`vector`	Not used (driver does not use interrupts).
`level`	Not used.
`performance`	Not used.

The default values for `sim1522` are:

```
struct adapter_info sim1522_info = {
    0,              /* Don't Care */
    0,              /* Controller ID */
    7,              /* SCSI ID of controller */
    0x340,          /* Controller address */
    0,              /* DMA channel */
    (32 + 11),      /* Interrupt vector */
    0               /* Don't care */
};
```

The 1522 is a very primitive chip with limited capabilities. The chip does not have a DMA controller like the 1542. To use DMA, the 1522 chip must use the DMA controller on the motherboard. This DMA controller is limited to 500 kilobytes using a 16 bit channel. Since it is faster to do programmed input/output (PIO) instead of DMA, DMA is not implemented for this driver.

SEE ALSO

Utility Programs - `fmtscsi`, `scandisk`

Device Facilities - `scsimgr`

NAME

`sim1542` – Adaptec 1540/1542 ISA SCSI device driver (x86)

DESCRIPTION

The SCSI device driver can control up to 7 devices on the SCSI bus. See the `scsimgr(4)` manual page for details on general devices supported. Target devices can be numbered 0 through 6; the CPU board controller is device 7.

Features of the `sim1542`:

- Uses `scsimgr`
- Bus tuning parameters allow for better disk performance.
- Support for disconnection/reconnection.
- Supports up to 3 cards in the system.

Limitations of the `sim1542`:

- ISA DMA see below.

The 1542 host adapter is limited with respect to DMA transfers to memory. It cannot transfer to physical memory greater than 16 megabytes. So on machines with more then 16 megabytes of memory the driver will DMA to memory below 16 megabytes then copy the data to the proper location. This only occurs when performing raw transfers and if the request contains memory greater than 16 megabytes. For example, if none of the memory to be transferred to is greater than 16 megabytes, no copy is performed.

The `info_structure` used by `sim1542` is defined as:

```
struct adapter_info {
    u_char scsi_port;      /* SCSI Bus/Port ID */
    u_char cntrl_id;       /* Controller ID */
    u_char scsi_id;        /* SCSI ID of host adapter */
    u_short base_address;  /* Adapter Base I/O address */
    u_short dma_chan;      /* DMA channel used */
    u_short vector;        /* Interrupt Vector */
    u_short level;         /* Interrupt Level */
    u_short bus_on;        /* bus_on time (usec) (2-15) */
    u_short bus_off;        /* bus_off time (usec) (1-64) */
};
```

The elements in this structure are described below.

`scsi_port`	Not used.
`cntrl_id`	Not used.
`scsi_id`	ID of host adapter (should be 7).
`base_address`	Port address of host adapter.
`dma_chan`	DMA channel used by host adapter.
`level`	Not used.
`vector`	Interrupt vector used by host adapter.
`bus_on`	How long to stay on bus during DMA transfer.
`bus_off`	How long to stay off bus after DMA transfer.

The default values for `sim1542` are:

```
struct adapter_info sim1542_info = {
    0,                /* Don't Care */
    0,                /* Controller ID */
    7,                /* SCSI ID of controller */
    0x330,            /* Controller address */
    5,                /* DMA channel */
    (32 + 11),        /* Interrupt vector */
    0,                /* Don't care */
    8,                /* bus_on time (2-15) */
    40                /* bus_off time (1-64) */
};
```

The 1542 can be tuned by setting `bus_on` and `bus_off` times in the `struct sim1542_info` contained in `sim1542_info.c`. These times are in micro-seconds. They represent the maximum amount of time the controller can stay on the bus during a DMA transfer and the minimum amount of time required to stay off the bus after performing a DMA transfer. The default settings for LynxOS are `bus_on`: 8, `bus_off`: 40. A maximum setting of `bus_on`: 15, `bus_off`: 1 can be used to boost disk performance. Setting these values can dramatically affect real time response due to DMA cycle stealing. Worst case response can double or triple. Lynx does its real time testing at the default settings.

SEE ALSO

Utility Programs - `fmtscsi`, `scandisk`

System Calls - `ioctl()`

Device Facilities - `scsimgr`

NAME

`sim1742` – Adaptec 1740/1742 EISA SCSI device driver (x86)

DESCRIPTION

The `sim1742` driver is an EISA SCSI driver. It requires the use of an EISA configuration disk. The SCSI device driver can control up to 7 devices on the SCSI bus. See the `scsimgr(4)` manual page for details on general devices supported. Target devices can be numbered 0 through 6; the CPU board controller is device 7.

Features of the `sim1742`:

- Uses `scsimgr`.
- Support for disconnection/reconnection.

The `info_structure` used by `sim1742` is defined as:

```
struct adapter_info {
    long vector;     /* IRQ + 32 */
    long iobase;     /* I/O Base Address */
    long bios;       /* BIOS Address */
    int  dma;        /* DMA channel */
    int slot;        /* EISA slot number */
    int cntrl_id;    /* Controller ID normally 7 */
    int scsi_id;     /* SCSI ID  */
    int scsi_port;   /* SCSI CAM port (assigned by scsimgr)
                       leave zero*/
    int int_type;    /* interrupt type LEVEL = 0, EDGE = 1 */
};
```

The fields in this structure are described below.

`vector`	Interrupt card uses.
`iobase`	Base address of host adapter.
`bios`	Not used.
`dma`	DMA channel used by host adapter, only necessary in 1542 emulation mode.

`slot`	Not used.*
`cntrl_id`	Not used.
`scsi_id`	Not used.
`scsi_port`	Not used.
`int_type`	Must be set to 1 (EDGE), does not matter in 1542 emulation mode.

*The `sim1742` automatically detects the `slot` that contains the card.

The 8259 is set up in edge trigger mode so the 1742 must set the `int_type` to `EDGE`. The `sim1742` driver does not auto-detect the parameters of the 1742 host adapter. Therefore, these values must be set in the device “info” file and the info file values must match values set in the EISA configuration program. The default values for `sim1742` are:

```
struct adapter_info sim1742_info = {
    (32 + 11),      /* Interrupt vector */
    0x330,          /* I/O base */
    0xDC000,        /* Not used.*/
    5,              /* DMA Channel */
    3,              /* Not used.*/
    0,              /* Controller ID  */
    7,              /* SCSI ID  */
    0,             /* SCSI CAM port (assigned by scsimgr)
                        leave zero*/
    1 /* 0=LEVEL,1=EDGE */
};
```

NOTE

This driver is an endangered species. It has been replaced by Adaptec with the 2742. It is very difficult to find a 1742 card. We support the 2742 and recommend the use of that driver instead of the `sim1742`.

SEE ALSO

Utility Programs -	`fmtscsi`, `scandisk`
System Calls -	`ioctl()`
Device Facilities -	`scsimgr`

NAME

`sim2742` – Adaptec 2740/2742 EISA SCSI device driver (x86)

DESCRIPTION

The `sim2742` driver is an EISA SCSI driver. It requires the use of an EISA configuration disk. The SCSI device driver can control up to 14 devices on the SCSI bus. See the `scsimgr(4)` manual page for details on general devices supported. Target devices can be numbered 0 through 14; the CPU board controller is device 7 (and 15 for the twin channel).

Features of the `sim2742`:

- Support for 2742 single channel host adapter.
- Support for 2742T twin channel host adapter.
- Uses `scsimgr`.
- Support for disconnection/reconnection.
- Supports 7 devices single channel.
- Supports 14 devices twin channel. (7 on each channel)

The `info_structure` used by `sim2742` is defined as:

```
typedef struct adapter_info {
    u_char  scsi_port;      /* SCSI Bus/Port ID */
    u_char  cntrl_id;       /* Controller ID */
    u_char  scsi_id;        /* SCSI ID of host adapter */
    u_short vector;         /* Interrupt Vector */
    u_short int_type;       /* Level triggered=0, Edge=1 */
    int slot;               /* EISA Slot number */
    int num_scsi_buses;     /* Don't set this at all */
} ADAPTER_INFO;
```

The elements in this structure are described below.

`scsi_port`	Not used.
`cntrl_id`	Not used.
`scsi_id`	ID of host adapter (should be set to 7).*

`vector`	Don’t care.
`int_type`	Not used.
`slot`	Slot that host adapter is in.
`num_scsi_buses`	This is used for internal purposes.

*Twin version of host adapter automatically sets 2nd channel ID of host adapter to 15.

The default values for `sim2742` are:

```
struct adapter_info sim2742_info = {
    0,              /* Don’t Care */
    0,              /* Controller ID */
    7,              /* SCSI ID of controller */
    (32 + 11),      /* Interrupt vector */
    0,              /* Don’t care */
    2               /* EISA slot # */
};
```

SEE ALSO

Utility Programs - `fmtscsi`, `scandisk`

System Calls - `ioctl()`

Device Facilities - `scsimgr`

NAME

`sim2940` – Adaptec 2940/2942 PCI SCSI device driver (x86)

DESCRIPTION

The SCSI device driver can control up to 15 devices on the SCSI bus. See the `scsimgr(4)` manual page for details on general devices supported. Target devices can be numbered 0 through 6, 8 through 15; the CPU board controller is device 7.

Features of the `sim2940`:

- Supports fast, wide, and ultra in all the legal combinations.
- Uses `scsimgr`
- Support for disconnection/reconnection.
- Up to 15 devices on the bus.
- Auto scans bus for the 2942 if `bus_num` and `dev_num` are set to 0.

The `info_structure` used by `sim2940` is defined as:

```
typedef struct adapter_info {
    u_char scsi_port;       /* SCSI Bus/Port ID */
    u_char cntrl_id;        /* Controller ID */
    u_char scsi_id;         /* SCSI ID of host adapter */
    u_short bus_num;        /* PCI Bus number */
    u_short dev_num;        /* PCI Device number */
} ADAPTER_INFO;
```

The elements in this structure are described below.

`scsi_port`	Not used.
`cntrl_id`	Not used.
`scsi_id`	ID of host adapter (Should be set to 7).
`bus_num`	*
`dev_num`	*

* If both fields are set to zero the `sim2940` driver will scan the bus for the host adapter. Otherwise these fields indicate where the card is located.

The default values for `sim2940` are:

```
struct adapter_info sim2940_info = {
    0,        /* Don't Care */
    0,        /* Controller ID */
    7,        /* SCSI ID of controller */
    0,        /* PCI Bus number */
    0         /* PCI Device number */
};
```

SEE ALSO

Utility Programs - `fmtscsi`, `scandisk`

System Calls - `ioctl()`

Device Facilities - `scsimgr`

NAME

`slip, SLIP` – serial line IP discipline

DESCRIPTION

SLIP is a line discipline which is used by networking software to encapsulate and transfer Internet Protocol (IP) datagrams over asynchronous RS-232 serial lines. A line discipline provides the semantics to control a serial device.

Functionality

SLIP is implemented in LynxOS via the `slattach` line discipline utility. This utility performs the serial port initialization, configuration, and it starts the communications line discipline. This provides the serial line with a network interface and prepares the system to transmit and receive network packets via the serial port. Each `SLIP` device is identified on the network by a unique network address. The `SLIP` network address and routing information are defined to the network by the `ifconfig` utility. The `network` can then route packets to and from a remote system connect via `SLIP`.

Installation

The following steps are required to configure a SLIP network interface:

1. Properly connect the two systems with a serial cable.
2. Modify the `/etc/hosts` files on both machines.
3. Disable the `login`s for the serial devices on both systems.
4. Run the `slattach` utility on both systems.
5. Run the `ifconfig` utility on both systems to set the network addresses.
6. Test the connection.

The following example and discussion are presented as a guide for the user to configure a simple `SLIP` connected network. The two systems diagrammed below, System-A and System-B, are to be connected together via a `SLIP` interface. System-B, in this example, is also acting as a gateway system to a larger network.

```
System A--------Serial Line Connection-----------System B
                193.1.3.1 (SLIP) 193.1.3.2       (Gateway)
                                                     |
                                                     |
                              Ethernet Connection  -
                              193.1.1.XXX
```

1. Connect the two systems together with a null modem type serial cable.
2. Modify the `/etc/host` files on both machine to include the network address of both systems. The network address for each system must be unique. For our example, we would make the following entries:

```
193.1.3.1 System-A System-A.lynx.com
193.1.3.2 System-B System-B.lynx.com
```

`Lynx.com` is the registered Internet name for Lynx Real-Time Systems. It is used here only as an example. Each site which connects to the Internet must have a unique name and network address. If you do not know yours, contact your company network administrator. They will probably provide you with both the system name and the network address to use for your configuration.

The serial ports being used to make the `SLIP` connection may have `login` sessions enabled. The `/etc/ttys` file on both system should be checked and edited as required to disable the `login`s on the `SLIP` serial ports. The status field of the tty entry should be a `0` for the login to be disabled. The format of the `/etc/ttys` entries is:

devicename:*status*:*baudrate*:*terminaltype*:*command*

For example:

```
/dev/com1:0:default:wyse50:/bin/login
```

If the `Status` entry is a `1`, this indicates that the device or port has been enabled for `login`. If the `Status` entry is a `0`, then the `login` has been disabled. If `login`s were enabled, the `login` process need to be terminated or the system reset.

3. The `slattach` utility needs to be run on both systems to configure the `SLIP` network interface. The arguments used by `slattach` are the interface device name and the interface device speed or baud rate. The following is an example to configure the `SLIP` device as `com1`:

```
slattach /dev/com1 9600
```

This configures the device `/dev/com1` as a `SLIP` interface using a communications speed of 9600 baud.

4. The `ifconfig` utility needs to be run on both systems to establish the network addresses and routing information for the systems. The arguments used by `ifconfig` are the dynamic device name, the local system's network address and the remote system's network address. The following is an example to configure the `SLIP` network address for System-A:

```
ifconfig sl0 193.1.3.1 193.1.3.2 link0
```

To configure System-B:

```
ifconfig sl0 193.1.3.2 193.1.3.1 link0
```

The option **`link0`** enables compression. For more information on **`link0`** options, please refer to the `ifconfig(1)` manual page.

The dynamic device name is determined by the `slattach` utility. The first dynamic device created by `slattach` is `sl0` (ess-ell-zero), the second is `sl1` (ess-ell-one). If a system has two `SLIP` devices installed the device that is configured by the second invocation of `slattach` has the dynamic device name of `sl1`.

Lastly, the `ping` utility should be used on both system to test the connection. The argument to the `ping` utility is the system name or network address of the remote system to connect with. The following is an example to test the `SLIP` connection from System-A to System-B:

```
ping 193.1.3.2
```

The `ping` utility will continue to run until interrupted by entering **Control-C** on System-A.

FILES

`/etc/tty`	File contains an entry for each terminal line on the system.

SEE ALSO

Utility Programs - `slattach, ifconfig, route`

System Calls - `ioctl(), fcntl()`

NOTES

If you are running `routed` from either of the nodes in the SLIP network, make sure to run the `routed` daemon from the other node as well. This ensures that the `routed` daemons can refresh the SLIP route information periodically. If `routed` is started on only one of the SLIP nodes, the `slip` routing entries within the kernel are subjected to the 180 sec timeout; in the absence of periodic routing information from the other node, the slip routing entries will be removed.

Rebooting the system may reset the SLIP configuration. If the interface configuration is needed permanently, it can be included in `/net/rc.network`.

To use the SLIP interface for NFS mounting, you must increase the NFS timeout value to 5 seconds (the default is 1 second). For example:

mount -o timeout=50 <*Slip_Remote_Server*>:<*Dir*> <*LocalDir*>

Please check the `mount(1)` manual page for more information on different NFS mount options.

NAME

`smc, dec` – interface to Ethernet (x86, PowerPC)

DESCRIPTION

The `smc/dec` device drivers are special character devices to perform communication over an Ethernet/IEEE802.3 data link.

The `smc` driver supports the SMC EtherPower card from Standard Micro Systems on x86 platforms. The `dec` driver supports the on-board DEC21040 Ethernet interface on PowerPC platforms.

The following `ioctl` functions are provided:

`SIOCSIFADDR`	Used by the `ifconfig` utility to pass-on the Internet address of the Ethernet interface.
`SIOCSIFFLAGS`	Used by the `ifconfig` utility to pass-on the flags to set/reset the interface.
`GE_SET_DEV`	Sets the interface in `raw_ethernet` mode. In this mode, the applications can directly communicate to the interface using `raw_ethernet` sockets.
`GE_SET_ADDR`	Changes the Ethernet address of the interface.
`GE_SET_PROTO`	Facilitates the setting up of the protocol range that the `raw_ethernet` socket needs to receive.
`GE_SET_MCAST`	Allows the user to set up a list of "multicast addresses" that the interface can receive as destination node using a `raw_ethernet` socket.
`GE_PROM`	When programming the interface to receive all packets, this `ioctl` utility determines whether or not it is meant for this node.
`GE_DEBUG_MODE`	Puts the interface in debug mode. The data transfers through this interface would be displayed in the debug terminal.
`GE_GET_CTRS`	Gets the `raw_ethernet` interface statistics.
`GE_GET_SADDR`	Gets the current Ethernet address of the interface.

GE_GET_FADDR	Gets the station address as assigned by the manufacturer of the ethernet interface hardware.
GE_CLRIFNAME	Closes the interface for raw_ethernet reception for a raw_ethernet socket.

The /sys/devices/if_dec21040info.c file has the following structure for configuring the Ethernet interface:

```
struct dcinfo {
    int xcvr_type ;
              /* Physical Ethernet connection:
                 0 = AUI/BNC; 1 = TP */
    int enmode;
                  /* mac address source: 0 = auto; 1 = manual */
    unsigned char enaddr[6];
              /* ethernet address if manual mode */
    int pci_bus;
              /* PCI bus number of the interface */
};
```

The xcvr_type field should be set to 0 for AUI, 1 for TP, 2 for 100 Mbit and 4 for BNC. The enmode field should normally be set to 0, unless you want to manually specify the Ethernet address in the enaddr field. The pci_bus field should be set to the PCI bus number where the adapter resides; it is normally bus 0.

FILES

/sys/devices/if_dec21040info.c

/sys/dheaders/if_dec21040info.h

/dev/smc (x86) and /dev/dec (PowerPC)

SEE ALSO

Utility Programs -	`ifconfig`
Device Facilities -	`raw_ethernet`
Documents -	See "Raw Ethernet Support" in the LynxOS User's Guide

NOTES

Unlike normal devices, this device cannot be operated with open/close `ioctl` calls directly. Instead, the device can be operated through sockets either in `AF_INET` domain or in `AF_RAWETH` domain.

NAME

`spx` – STREAMS pipe driver

DESCRIPTION

A stream pipe provides full-duplex connection between two file descriptors. It can be used for passing file descriptors between processes. The stream pipe driver is implemented as a clonable device driver. The number of stream pipe devices in a system can be configured by changing the streams tunable parameter `MAX_SPIPE` in the `/sys/devices/streams_info.c` file.

To create a pipe, the device `/dev/spx` has to be opened twice, getting two file descriptors `fd0` and fd1, and then using the stream `ioctl` command `I_FDINSERT` to connect the two file descriptors together. When the last user of one of the file descriptors closes that file descriptor, the streams pipe driver sends a `M_HANGUP` message to the stream head at the other end of the pipe. At the other file descriptor, `putmsg` system call will return with `ENXIO` error and the `poll` system call will return with `POLLHUP` event.

Using the `fstat` system call for one of the file descriptors, it is possible to get the device number associated with that file descriptor (the two file descriptors will have the same major device number, but different minor device numbers). This device number can be used with a `mknod` system call to create a character device node in the file system which is associated with the stream head for that file descriptor.

FILES

`/dev/spx`

SEE ALSO

System Calls - `putmsg(), poll()`

Device Facilities - `clone, streamio`

NAME

`streamio` – STREAMS ioctl commands

SYNOPSIS

```
#include <stropts.h>

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

DESCRIPTION

Control functions on STREAMS are preformed by a subset of the `ioctl` commands; consult the `ioctl` manual page in the system call section. The call sends the value of **`request`** and the pointer **`arg`** to `stream` associated with the file descriptor **`fd`** where they are interpreted by the `stream head.` A module or driver in the `stream` can also accept some of these arguments.

An open file descriptor that refers to a `stream` is passed in **`fd`**. The control function to be performed is determined by the value of **`command`**. If additional information is required by the value in **`command`** it is passed in **`arg`**.

In addition to the errors described in the **`ioctl`** manual page **`ioctl`** can fail with **`errno`** set to `EINVAL` without processing the control function. This happens if the command is not a valid value for the `stream`, or if the `stream` is located below a multiplexor.

STREAMS modules and drivers can also detect errors. When they do an error value is sent to the `stream head`. Future system calls will fail and **`errno`** will be set to the error value sent back to the `stream head`.

IOCTLS

All STREAMS files support the following **`ioctl`** commands:

I_PUSH

Pushes a module onto the top of the current `stream`, just below the `stream head` and then calls the open routine of the module. The name of the module is pointed to by **arg**. `I_PUSH` can fail with the following errors:

EINVAL	Invalid module name.
EFAULT	**arg** contained an illegal address.
ENXIO	The open routine failed.
ENXIO	Hangup was received on the `stream`.

I_POP

Pops the module just below the `stream head` off of the `stream`. `I_POP` can fail with the following errors:

EINVAL	No module was on the `stream`.
ENXIO	Hangup was received on the `stream`.

I_LOOK

Places the name of the module just below the `stream head` in a character buffer `FMNAMESZ+1` bytes long that is pointed at by **arg**. The declaration `#include <sys/conf.h>` is required. `I_LOOK` can fail with the following errors:

EINVAL	No module was present on the `stream`.
EFAULT	**arg** contained an illegal address.

I_FLUSH

Depending on the value of **arg**, flush all input and/or output queues. **arg** values can be:

FLUSHR	Flush read queues.
FLUSHW	Flush write queues.
FLUSHRW	Flush read and write queues.

`I_FLUSH` can fail with the following errors:

EAGAIN	Buffers for the flush message could not be allocated.
EINVAL	The value of **arg** was invalid.
ENXIO	Hangup was received on the `stream`.

`I_SETSIG` Informs the `stream head` to issue the `SIGPOLL` signal (consult the `sigvec` manual page) when the event specified in **`arg`** has occurred on the `stream`. In `STREAMS` `I_SETSIG` supports asynchronous processing. The event for which the signal should be sent is specified by a bitmask in **`arg`**. **`arg`** is built from the `bitwise_OR` of any combination of the following constants:

`S_INPUT` A non-priority message of zero or greater length has arrived on an empty `stream head` read queue.

`S_HIPRI` A priority message of zero or greater length has arrived on the `stream head` read queue.

`S_OUTPUT` Space is now available on the `stream head` write queue for sending data downstream.

`S_MSG` A message containing the `SIGPOLL` signal is at the top of the `stream head` read queue.

A user will only be signaled for priority messages if `S_HIPRI` is set in the **`arg`** bitmask.

Each process that wishes to receive `SIGPOLL` signals must register to receive them with `I_SETSIG`. Every process registered to receive the `SIGPOLL` signal for the same event on the same `stream`, will be signaled when the event occurs.

For a process to be unregistered and to no longer receive `SIGPOLL` signals, **`ioctl`** is called with **`request`** set to `I_SIGSET` and **`arg`** set to 0.

`I_SETSIG` can fail with the following errors:

`EINVAL` The value of **`arg`** is invalid.

`EAGAIN` No space available to store the signal request data structure.

`I_GETSIG` Returns to the calling process a bitmask indicating those events that the calling process is currently registered to be informed of with a `SIGPOLL` signal. The bitmask is pointed to by **arg**. The events are described in `I_SETSIG`. `I_GETSIG` can fail with the following errors:

`EINVAL` The calling process is not registered with `I_SETSIG`.

`EFAULT` **arg** contained an illegal address.

`I_FIND` Looks for a specified module in the `stream`, returning 1 if the module is found and 0 if it is not. The name of the specified module is pointed to by **arg**. `I_FIND` can fail with the following errors:

`EFAULT` **arg** contained an illegal address.

`EINVAL` **arg** points to an invalid module name.

`I_PEEK` Retrieve the information in the first message on the `stream head` read queue and leave the message on the queue. **arg** points to a `strpeek` structure which looks like:

```
struct strbuf ctlbuf;
struct strbuf databuf;
long flags;
```

The length of the control information and data information to be retrieved is specified in the `maxlen` field of `ctlbuf` and `databuf` `strbuf` respectively.

To only look for a priority message on the `stream head` read queue the flags field is set to `RS_HIPRI`.

If a message is retrieved then `I_PEEK` returns 1; otherwise, `I_PEEK` returns 0. If there is no message present `I_PEEK` does not wait. On return, control information is returned in `ctlbuf`, data is returned in `databuf` specifies, and `flags` contains either 0 or `RS_HIPRI`.

`I_PEEK` can fail with the following errors:

`EFAULT` **arg**, `ctlbuf` or `databuf` contained an illegal address.

`I_SRDOPT` Set the read mode to the value specified in **arg**. Legal values for **arg** are:

`RNORM` Byte-stream mode.

`RMSGD` Message-discard mode.

`RMSGN` Message-nondiscard mode.

`RNORM` is the default. More information on read modes can be found in the `read` manual page in System Call section of User's Manual, Volume 1. `I_SRDOPT` can fail with the following error:

`EINVAL` **arg** is not a legal value.

`I_GRDOPT` Returns the current read mode in an **int** pointed to by the **arg**. `I_GRDOPT` can fail with the following error:

`EFAULT` **arg** contained an illegal address.

`I_NREAD` Returns the number of data bytes in the first message on the `stream head` read queue in the location pointed to by **arg**. **ioctl** also returns the number of messages on the `stream head` read queue. For example, if **ioctl** returns 10 and zero is returned in **arg**, then there are 10 messages in the stream and the first message is of length zero. `I_NREAD` can fail with the following error:

`EFAULT` **arg** contained an illegal address.

`I_FDINSERT` Builds a message and sends it downstream. The message is built from user defined control information, optional user defined data and additional information about another stream.

arg points to a `strfdinsert` structure which looks like:

```
struct strbuf ctlbuf;
struct strbuf databuf;
long flags;
int fd;
int offset;
```

`ctlbuf` is a structure that contains the user defined control information. The size of a pointer plus the number of bytes of control information to be sent with the message is passed in the `len` field of `ctlbuf`.

`databuf` is a structure that contains the user defined data. The number of bytes of data information to be sent with the message or zero if no data part is to be sent is passed in the `len` field in `databuf`. The file descriptor of the other stream is passed in `fd`. The number of bytes beyond the start of the control buffer, where `I_FDINSERT` will store a pointer to the other stream's driver read queue structure, is passed in `offset`. `offset` must be word-aligned.

If `flags` is set to 0 a non-priority message is created, and if `flags` is set to `RS_HIPRI` a priority message is created. `I_FDINSERT` will block when the `stream` write queue cannot accept a message due to internal flow control conditions, and if it is a non-priority message and `O_NDELAY` is not set. Internal flow conditions will cause priority messages to fail and set **`errno`** to `EAGAIN`.

Regardless of priority or `O_NDELAY`, `I_FDINSERT` will block waiting for message blocks in the `stream`. No partial message will be sent.

`I_FDINSERT` can fail with the following errors:

`EAGAIN`	The `stream` write queue is full due to internal flow control conditions and the command could not block.
`EAGAIN`	No memory available for the buffers required by the message.
`EFAULT`	**`arg`**, `ctlbuf`, or `databuf` contained an illegal address.
`EINVAL`	`fd` in the `strfdinsert` structure is not a valid.
`EINVAL`	The size of the pointer plus `offset` is greater than the `len` field in `ctlptr`.
`EINVAL`	`offset` does not specify a word-aligned location in the data buffer.
`EINVAL`	`flags` points to an undefined value.
`ENXIO`	A hangup is received on the `stream`.

ERANGE The len field in databuf is out of range for the top-most stream module.

ERANGE The len field in databuf is larger than the maximum allowed size for the data part of a message, or the len field in ctlbuf is larger than the maximum allowed size for the control part of a message.

I_STR Build an internal STREAMS **ioctl** message and send that message downstream. The message is built from the data pointed to by **arg**.

I_STR is used to send **ioctl** requests, variable amounts of data, and timeout specifications to downstream modules and drivers. It returns any information sent upstream by the downstream module or driver. I_STR blocks until the downstream recipient responds or until the call “times out”. For a “time out” **errno** is set to ETIME.

Only one I_STR can be active on a stream. Any other I_STR call will block until the active I_STR completes.

arg must point to a strioctl structure which looks like:

```
int ic_cmd;        /* downstream command*/
int ic_timout;     /* ACK/NAK timeout*/
int ic_len;        /* length of data arg*/
char *ic_dp;        /* ptr to data arg*/
```

The **ioctl** command that is to be sent to the downstream driver is passed in the ic_cmd field. The number of seconds to wait before timing out is passed in ic_timout. If ic_timout is -1 then I_STR will wait forever. If ic_timout is 0 then I_STR will wait the default time, 15 seconds. The number of bytes in the data argument is passed in the ic_len field. A pointer to the data argument is passed in ic_dp. Both ic_len and ic_dp are used to send data to and receive data from the downstream module or driver. The buffer pointed to by ic_dp must be able to contain as much data as the downstream module or the driver can return).

The information in the strioctl structure is converted to an internal **ioctl** command message and sent downstream by the stream head.

I_STR can fail with the following errors:

EAGAIN	Could not allocate buffers for the **ioctl** message.
EFAULT	**arg** or ic_dp contained an illegal address, or ic_db + ic_len is an illegal address.
EINVAL	ic_len is less than 0 or greater than the maximum size of the data part of a message.
EINVAL	ic_timout is less than -1.
ENXIO	A hangup is received on the stream.
ETIME	The down stream **ioctl** command timed out.

If an error or a hangup message is received at the stream head while I_STR is waiting I_STR will fail. If the downstream **ioctl** command fails, I_STR will fail and **errno** will be set according to the downstream failure.

I_SENDFD
: Sends a message containing a file pointer to the stream head at the other end of the stream. **arg** contains a file descriptor which will be converted into the file pointer that will be sent.

I_SENDFD allocates a message block then inserts the file pointer, and the user and group id associated with the sending process are into the message block. This message is placed on the read queue of the stream head at the other end of the stream.

I_SENDFD can fail with the following errors:

EAGAIN	Could not allocate message block.
EAGAIN	The receiving stream head read queue is full.
EBADF	**arg** is not a valid.

EINVAL	**fd** is not a valid file descriptor.
ENXIO	A hangup is received on the stream.

I_RECVFD

Retrieves the file descriptor associated with a message sent by I_SENDFD. **arg** A data buffer large enough to hold a strrecvfd data structure is pointed to by **arg**. The strrecvfd structure looks like:

```
int fd;
unsigned short uid;
unsigned short gid;
char fill[8];
```

fd is a file descriptor. uid and gid are the user ID and group ID of the sending stream.

I_RECVFD will block until a message is present at the stream head unless O_NDELAY is set. If no message is present at the stream head and O_NDELAY is set then I_RECVFD will fail with **errno** set to EAGAIN.

I_RECVFD converts the file pointer sent by I_SENDFD into a file descriptor. This new file descriptor is placed in the fd field of the strrecvfd structure. The structure is then copied into the user data buffer pointed to by **arg**.

I_RECVFD can fail with the following errors:

EAGAIN	The O_NDELAY flag is set and there is no message present at the stream head read queue.
EBADMSG	The message at the stream head read queue was not sent by I_SENDFD.
EFAULT	**arg** contained an illegal address.
EMFILE	Too many descriptors are active.
ENXIO	A hangup is received on the stream.

The following commands are used to connect and disconnect multiplexed STREAMS configurations.

I_LINK

Connect two streams. **fd** is the file descriptor of a stream connected to a multiplexing driver. **arg** is the file descriptor of the stream connected to another driver. The stream designated by **arg** gets connected below the multiplexing driver. The multiplexing driver sends an acknowledgment message to the stream head about the connection. I_LINK returns a multiplexor ID number on success, and a -1 on failure.

I_LINK can fail with the following errors:

ENXIO	A hangup is received on the stream.
ETIME	The **ioctl** timed out.
EAGAIN	Could not allocated memory required by I_LINK.
EBADF	**arg** is not valid.
EINVAL	Multiplexing not supported by the stream referred to by **fd**.
EINVAL	**arg** is not a stream, or is already lined under a multiplexor.
EINVAL	The specified link would cause a stream head to be linked into a multiplexing configuration in more than one place.

If an error or a hangup message is received at the stream head while I_LINK is waiting I_LINK will fail. Also an error code can be returned in the acknowledgment message. **errno** will be set to the returned error code.

I_UNLINK

Disconnect the two streams. **fd** is the file descriptor of the stream connected to the multiplexing driver. **arg** is the multiplexor ID number returned by I_LINK when the stream was linked below the multiplexing driver. All streams which were linked to **fd** are disconnected if **arg** is set to -1. The multiplexing driver must acknowledge the unlink.

I_UNLINK can fail with the following errors:

ENXIO	A hangup is received on the stream.

`ETIME` The **`ioctl`** timed out.

`EAGAIN` Could not allocate buffers for the acknowledgment message.

`EINVAL` Invalid multiplexor ID number.

If an error or a hangup message is received at the `stream head` while `I_UNLINK` is waiting, `I_UNLINK` will fail. Also, an error code can be returned in the acknowledgment message. **`errno`** will be set to the returned error code.

SEE ALSO

System Calls - `close(), fcntl(), intro, ioctl(), open(), read(), sigvec(), write()`

Calls for STREAMS special files include:

System Calls- `getmsg(), poll(), putmsg()`

NAME

`tcp`, `TCP` – Internet Transmission Control Protocol

SYNOPSIS

In the socket environment:

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

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

In the TLI environment:

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

t = t_open("/dev/tcp", O_RDWR, info);
```

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 in the socket environment and the `T_COT` type of transport service in the TLI environment. `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 may use the `accept(2)` call to accept incoming connections. Only active sockets may use the `connect(2)` call to initiate connections.

In the TLI environment the device `/dev/tcp` is used to implement the TCP/IP protocol. The device `/dev/tcp` is a STREAMS clonable device driver. The number of minor devices for this driver is specified by the parameter SMD_NTCP in the /sys/devices/stsocket_info.c file. After opening /dev/tcp, the transport-end point can be specified to be passive by passing a non-zero value in the qlen field in the t_bind structure of a t_bind(3) TLI library function. Only passive endpoints may use the t_accept(3) call of the TLI library function. Both passive and active transport endpoints may initiate a connection using the t_connect(3) call.

Passive sockets may 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 which listens on all networks, the Internet address INADDR_ANY must be bound. The TCP port may 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 to 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 which 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 acknowledgment is received. For a small number of clients, such as window systems that send a stream of mouse events which receive no replies, this packetization may cause significant delays. Therefore, TCP provides a boolean option, TCP_NODELAY (from <netinet/tcp.h>), to defeat this algorithm. The option level for the setsockopt call is the protocol number for TCP, available from getprotobyname(3).

Options at the IP transport level may be used with TCP; see ip(4). Incoming connection requests that are source-routed are noted, and the reverse source route is used in responding.

In the TLI environment, currently no options can be set.

DIAGNOSTICS

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

`EISCONN`	When trying to establish a connection on a socket which already has one.
`ENOBUFS`	When the system runs out of memory for an internal data structure.
`ETIMEDOUT`	When a connection was dropped due to excessive retransmissions.
`ECONNRESET`	When the remote peer forces the connection to be closed.
`ECONNREFUSED`	When the remote peer actively refuses connection establishment (usually because no process is listening to the port).
`EADDRINUSE`	When an attempt is made to create a socket with a network address for which no network interface exists.
`EADDRNOTAVAIL`	When an attempt is made to create a socket with a network address for which no network interface exists.

SEE ALSO

System Calls - `getsockopt(), socket()`

Library Functions - `t_open(), t_bind(), t_listen(), t_accept(), t_connect()`

Device Facilities - `intro, inet, ip`

NOTES

It should be possible to send and receive `TCP` options.

NAME

`tty` – hardcopy and video terminal interface

SYNOPSIS

#include <sgtty.h>

#include <ioctl.h>

#include <termio.h>

DESCRIPTION

This section describes the device driver responsible for conversational computing. This device driver is often referred to as the `tty` driver and is used to facilitate communication between different kinds of terminals and the LynxOS host. In describing various ASCII characters, the convention of using a “`^`” character before a character will be used to denote a control character. For instance: “`^B`” means control-B. In addition, all variables referred to in the following discussions about modes and functions are kept in the main structure described in the section on Operational Parameters.

A unique feature of the `tty` driver is that most of the keys which control various driver actions (delete word, backspace, etc) are soft; that is, most driver actions have a variable associated with them. The variable contains the ASCII character which causes that particular action to occur whenever it is typed. The variable can be set to any value with `ioctl` calls. To disable a particular action, the variable for that action can simply be set to a character which can never be typed, i.e. `-1`.

Operational Modes

The `tty` driver has three major modes of operation: cooked, cbreak, and raw. These modes are characterized by the amount of processing they perform on input and output. A fourth mode, System V non-canonical mode, is identical to `raw` mode except in the way input processing is handled. Some of the terms mentioned in the following description of the different modes are explained in separate sections as they are common to more than one mode.

Cooked Mode

The majority of input/output processing is performed in cooked mode. It is most commonly used for interaction with software because it has facilities for line editing, interrupt signal generation, modem control, and several forms of output processing.

To allow line editing, the textual unit of input in this mode is an entire "line" of text, where a "line" is a sequence of characters terminated by a line termination character such as CR (carriage return) or `^D`. Thus, characters typed into a terminal in this mode are not immediately made available to the calling program. They are first buffered to allow the user to perform various line editing functions on them. The buffer into which the characters are placed is 256 bytes long. If this buffer becomes full, the `tty` driver will ignore any new input. Even the line termination character will be ignored.

The only characters that will be accepted will be editing characters that would result in a reduction of the number of characters in the buffer.

The following is a description of the cooked mode line editing functions:

`erase character`
: Invoked by typing the character in either the `sg_erase` or `sg2_erase` variables. This function is used to erase single characters on a line. The default value of `sg_erase` is the "backspace" character `^H`, and the default value of `sg_2erase` is the "delete" character (ASCII code 127). This function is directly affected by the `LCRTBS` and `LPRTERA` flags. (these flags are documented below in the description of the local mode word).

`erase word`
: Selected by typing the character in the `sg_werasc` variable. This function is used to erase entire "words", where a word is a series of characters delimited by spaces. The default value of `sgw_erasc` is `^W`. This function is directly affected by the following flags: `LCRTBS`, `LPRTERA` (these flags are documented below in the description of the local mode word).

`erase line`
: Selected by typing the character in the `sg_kill` variable. This function is used to erase the entire line. The default value of `sg_kill` is `^U`. This function is directly affected by the following flags: `LCRTBS`, `LPRTERA`, `LCRTKIL` (these flags are documented below in the description of the local mode word).

`reprint line`
: Selected by typing the character in the `t_rprntc` variable. This function is used to reprint the entire line. This may be necessary when extraneous characters are sent to the user's terminal from some foreign source (i.e., output from some background process). The default value of `t_rprntc` is `^R`.

`end of file`
: Selected by typing the character in the `t_eofc` variable. This function makes an input line available to any programs reading from the current terminal, and causes the next read request to return 0 (`EOF`).

`delimit line`
: Selected by typing the character in the `t_brkc` variable. This function is used to terminate a line of input. If the `CRMOD` flag is enabled, the driver echoes carriage return and line feed; otherwise, the character is echoed directly. The default value of `t_brkc` is -1 (disabled).

In addition to the line editing features described above, cooked mode also supports signal generation, modem control, and various forms of input and output processing, but because some of these features are also available in cbreak and raw modes they are described separately.

cbreak Mode

`cbreak` mode supports all of the features of cooked mode except for those pertaining to line editing. This implies that it supports output processing, signal generation, input control, flow control, and modem control. Since there is no line editing available, input is made available to the calling program as soon as it is received.

Raw Mode

This mode contains the least number of features of all the modes. It is used primarily for applications requiring a minimum of processing overhead.

The only signal which can be generated from the keyboard in raw mode is the `SIGBRK` signal. This signal is sent every time a break condition is detected on the line. This condition can usually be set by typing the `BREAK` key on the terminal's keyboard. A break condition is reflected in the input stream as a null character.

Characters read and written in raw mode are processed as 8-bit characters. This is in contrast to the other modes in which all characters are stripped of the eighth bit. Also, because no line editing is available, input is returned to the calling program as soon as it is requested. Unless `TANDEM` mode is in effect, any characters received after input queue overflow are discarded.

System V Non-Canonical Mode

The LynxOS `tty` driver supports System V non-canonical mode input processing. To enter this mode, the System V-compatible `TCSETA` `ioctl` call must be used with the `ISIG` and `ICANON` bits in the `c_lflag` field of the `termio` structure turned off. When the `tty` driver is placed in System V non-canonical mode, the fifth and sixth control characters (i.e. `c_cc[4]` and `c_cc[5]`) of the `termio` structure take on the meanings of `MIN` and `TIME`, respectively. The `MIN` and `TIME` values dictate how input characters are processed. In general, a read request in System V non-canonical mode will not return until `MIN` characters have been received or `TIME` tenths of a second have elapsed. The point at which the timer begins depends on whether or not `MIN` is zero. There are four distinct behaviors depending on the values of `MIN` and `TIME`:

`MIN>0, TIME>0`

In this case, `TIME` represents an inter- character timer and is not activated until the first character is received. A read request will not return until `MIN` characters have been received or `TIME` tenths of a second have elapsed between the receipt of two successive characters, whichever comes first. The timer is reset upon receipt of each character. A read will block indefinitely until the first character is received.

`MIN>0, TIME=0`
: In this case, there is no timer. A read request will not return until `MIN` characters have been received.

`MIN=0, TIME>0`
: In this case, `TIME` represents a request timer and is activated as soon as the read request is made. A read request will return as soon as a character is received or `TIME` tenths of a second have elapsed since the read request was made, whichever comes first. Since the timer is activated on the read and not on each received character, it is possible in this case for the read to return zero characters.

`MIN=0, TIME=0`
: In this case, a read request always returns immediately. The number of characters returned will be the minimum of the number requested and the number currently available, which could be zero.

It is important to realize that `MIN` does not serve as a maximum. For example, if `MIN` is 5 and there are 10 characters available, a read request for 7 characters will return 7 characters (not 5).

Flow Control

Flow control is available to all modes in some degree. Start/stop output flow control is available only in cooked and cbreak modes. Outgoing flow control is available in all modes. The following is a summary of flow control functions:

`stop output`
: Selected by typing the character in the `t_stopc` This function disables output until the character in the `t_startc` variable is typed. The default value of `t_stopc` is `^S`. If the value in `t_stopc` is the same as the value in `t_startc` a toggle effect can be achieved. This function works in cooked and cbreak modes only.

`start output`
: Selected by typing the character in the `t_startc` variable. This function enables output. It is used to start output after it has been disabled by the stop output

function. If the LDECCTQ flag in the local mode word is not set, then any character received while output is stopped resumes output.

`tandem flow control`
: This option is set by setting the TANDEM flag in sg_flags. It causes the tty driver to send a stop character to the device at the other end of the line (your terminal, another computer, etc) whenever its input queue is in danger of overflowing. A start character is sent when the input queue has drained sufficiently and new input can be accepted. This mode is useful when communicating with another computer. This function works in all modes.

Output Processing

Output processing is performed to some degree in all modes. Since it takes a rather long time (relative to the speed at which the CPU operates) to output a character to a device such as a terminal, the tty driver puts all of its output into a queue. This queue is in turn drained by an interrupt routine activated by the output hardware. If a process fills up the queue faster than it can be drained, the tty driver will suspend the process until the queue has drained enough that it can be written to again. Characters are normally output with no parity and 1 stop bit, unless information in sg_flags dictates otherwise.

Output Of The EOT Character

Since it is possible for some terminals to hang up upon receipt of the end-of-transmission character (EOT), it is not echoed or output when the tty driver is in cooked mode. EOT is however, output and/or echoed in both raw and cbreak modes, so programs using these modes should be careful.

Case Mapping

The tty driver has a provision for mapping upper-case to lower-case for terminals which only have lower-case. This option is selected by setting the LCASE bit in the sg_flags word.

Output Flushing

To suppress extraneous output, the `tty` driver supports output flushing with the `t_flushc` variable. Typing this character causes the `tty` driver to continuously flush its output queue until more input is typed. Upon receipt of this character, bytes to be written are instead ignored until any other character is received. The default value of `t_flushc` is `^O`. This function is only available in cooked or cbreak modes.

Signal Generation

Signal generation is supported in all modes to some degree. All signals sent by the `tty` driver are sent to the processes in the current process control group. The processes receiving the signals receive them as if they were sent to them via the `killpg` system call. What follows is a description of all the signals that the `tty` driver can generate along with a description of the ways that they can be sent:

`SIGBRK` — This signal is sent whenever a break condition is detected by the input hardware. This condition may be detected in the input as a null character in raw mode or as the interrupt character in cooked or cbreak modes. The break condition can be set manually by depressing the `BREAK` key on a terminal.

`SIGHUP` — This signal is sent when the carrier signal from the dataset drops. This usually happens when a user has hung up a modem. The driver sends this signal to the entire process group. Unless caught or ignored, `SIGHUP` causes a process to terminate. Transmission of this signal can be suppressed by setting the `LNOHANG` bit in the `local mode word`.

`SIGINT` — This signal is usually sent to stop a process or to regain control of a program that is running interactively. It is sent by the driver to all the processes in the current `tty` process group upon receipt of the character contained in `t_intrc`. The value of this variable is usually `^C`.

`SIGIO` — When enabled, this signal is sent to all processes in the tty process group whenever input is available to be read. Transmission of the signal is permitted when the `fcntl` system call is sent to a tty device with the `F_SETFL` argument and the flags set to FASYNC. Transmission of the signal is disabled by clearing the FASYNC flag in the same fcntl system call. Transmission of SIGIO can also be enabled with calling ioctl with the TIOASYNON argument. Transmission of SIGIO can be disabled by calling ioctl with the TIOASYNOFF argument.

SIGQUIT
: This signal is usually used to terminate a program and cause it to produce a core dump in the current working directory. It is sent upon receipt of the character contained in the t_quitc. The default value of this variable is ^\.

SIGTSTP
: This signal is used to suspend the processes in the current process group. It is sent upon receipt of the character contained in either the t_suspc or the t_dsusp variables. In the case of t_dsusp, the transmission of this signal is delayed until the character is actually read by a process from the tty drivers's input queue. The default value of t_suspc is ^Z. The default value of d_suspc is ^Y.

SIGTTIN
: This signal is sent to processes which attempt to read from the tty but do not belong to the tty process group. Normally, this signal stops the receiving process. If the signal is ignored, or the process is an orphan, the read returns 0.

SIGTTOU
: The transmission of SIGTTOU is enabled by the LTOSTOP bit. This signal is sent to processes which do not belong to the process group of the tty but attempt write operations. If these processes are ignoring the SIGTTOU signal, or are orphans, they are allowed to produce output.

Driver Operation Modification Flags

In order to change modes, and perform other operational modifications, there exists a set of flags. This set of flags is contained in the variable sg_flags. These flags can be set and cleared with the ioctl calls TIOCSETP, TIOCGETP, and TIOCSETN. A further description of these flags follows:

XTABS	Replace tabs with the appropriate number of spaces on output. The number of spaces per tab is determined by the `tab_width` variable.
EVENP	Set even parity (input and output).
ODDP	Set odd parity (input and output).
CRMOD	Setting this flag causes the carriage return character (`CR`) to be mapped on input to the linefeed character (`LF`). It also causes `CR` to echo as `CR-LF` (carriage return, linefeed).
ECHO	`echo` characters received (full-duplex).
LCASE	Causes the driver to interpret upper case alphabetic characters as their lower case equivalents. Lower case characters are also understood to be lower case, but are echoed as upper case. Upper case characters which are written to the port are printed with a preceding backslash. Additionally, certain punctuation characters are echoed and written differently:

Character	Echoed As
{	\ (
}	\ )
|	\ !
~	\ ^
‘	\ ’

RAW	Selects raw mode.
CBREAK	Selects `cbreak` mode. If both `RAW` and `CBREAK` are set or cleared, cooked mode is selected.
TANDEM	Enables outgoing flow control. Most often used when communicating with another computer over a serial port.

The Local Mode Word

The local mode word contains flags which along with `sg_flags` select driver options. It is contained in the variable `l_localmode` and can be directly manipulated with the `ioctl` calls `TIOCLBIS`, `TIOCLBIC`, `TIOCLSET`, and `TIOCLGET`. The `TIOCSETP`, `TIOCGETP`, and `TIOCSETN` calls may also be used. The following are flags in the local mode word that affect the operation of both cooked and cbreak modes of the `tty` driver:

`LCRTBS` — This bit controls an aspect of how the `tty` driver erases characters from a line. If set, the driver writes the character erase sequence every time the `erase character` is typed (`sg_erase`). If cleared, the driver simply echoes the `erase character` itself. If `sg_erase` is written directly, an output translation is performed on it based on the state of `LCTLECH`.

`LPRTERA` — This bit controls the selection of printing terminal erase mode. This feature is useful for hardcopy terminals which are incapable of erasing characters from their display surface. Setting this bit causes a "\"to be printed followed by the erased characters echoed in the order that they are logically erased. To denote the end of an erase sequence, the echoed characters are followed by a "\" character.

`LCRTERA` — This bit controls the selection of crt erase mode. When set, it causes a "backspace-space-backspace" sequence of character to be output to the screen to erase a character. If cleared, just the character in `sg_erase` is echoed.

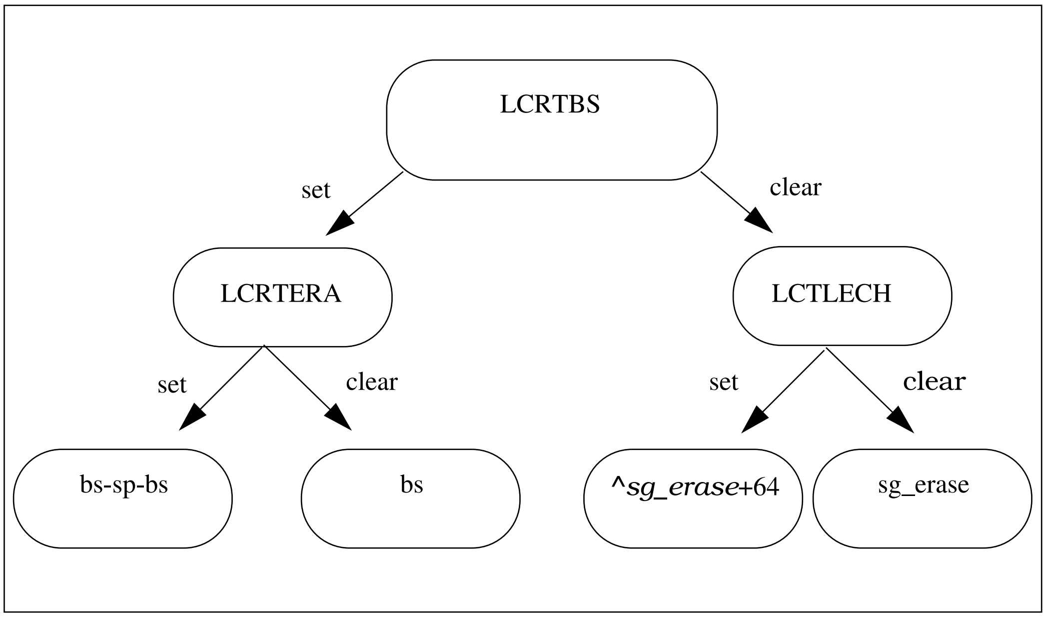
`LMDMBUF` — When set, this bit toggles the output start/stop when the carrier drops. When cleared, output simply stops when the carrier drops.

`LLITOUT` — Inhibits all output translations when set.

`LTOSTOP` — If set, then any process in a process group other than the port process group which writes to the port will be sent a `SIGTTOU` signal.

`LFLUSHO`	When set, this bit indicates that the output queue is currently being flushed. After the queue has been flushed, this bit is cleared.
`LNOHANG`	When set, this bit inhibits the transmission of a hang-up when the carrier signal from the dataset drops.
`LCRTKIL`	When set, this bit causes multiple character erase sequences to be sent to erase an entire line in response to a line erase request. When clear, just the character in `sg_kill` and a carriage return sequence are echoed to indicate that the line has been erased internally.
`LNOMDM`	When set, the driver ignores all modem control signals.
`LCTLECH`	When set, the driver echoes control characters as a “`^`” followed by the printable character corresponding to the control character. For example, control-A would be echoed “`^A`”.
`LPENDIN`	An internal state bit which indicates that characters were pending during a switch from `cbreak` to `cooked` mode, and will be re-input.
`LDECCTQ`	When cleared, it modifies the operation of the start output function so that output can be restarted by any character typed rather than just the character in the `t_startc` variable. When set, only the character in the `t_startc` variable can restart output (as with DEC operating systems).

The following diagram explains how LCRTBS, LCRTERA, and LCTLECH interact with each other in terms of what is echoed when the sg_erase character is typed:



Operational Parameters

The user-modifiable operational parameters of the tty driver are accessed via various ioctl calls and are contained in the sgttyb, tchars, and ltchars structures. The sgttyb is set and retrieved using the TIOCSETP and TIOCGETP ioctls; tchars is set and retrieved using TIOCSETC and TIOCGETC; and struct ltchars is set and retrieved using TIOCSLTC and TIOCGLTC. In addition, the local mode word of the tty can be retrieved with TIOCLGET, and set with TIOCLSET; moreover, you can set bits in it by using TIOCLBIS and clear bits by using TIOCLBIC. All ioctl requests are more fully described below. The structures are defined in **ioctl.h** as follows with defaults shown in ():

```
struct sgttyb {
    char sg_ispeed;      /* input baud rate */
    char sg_ospeed;      /* output baud rate */
    char sg_erase;       /* erase char (default "#") */
```

```
        char sg_2erase; /* 2nd erase character */
        char sg_kill;   /* kill char ("@") */
        int  sg_flags;  /* various modes and delays flag */
    };
    struct tchars {
        char t_intrc;  /* interrupt char ("^?") */
        char t_quitc;  /* quit char ("^\") */
        char t_startc; /* start output ("^Q") */
        char t_stopc;  /* stop output ("^S") */
        char t_eofc;   /* end-of-file char ("^D")*/
        char t_brkc;   /* input delimiter like nl,( -1 ) */
    };
    struct ltchars {
        char t_suspc;   /* stop process signal ("^Z") */
        char t_dsuspc;  /* delayed stop proc signal ("^Y")*/
        char t_rprntc;  /* reprint line char ("^R") */
        char t_flushc;  /* flush output toggles ("^O") */
        char t_werasc;  /* word erase character ("^W") */
        char t_lnextc;  /* literal next character ("^V") */
    };
```

struct sgttyb

`sg_ispeed` Contains the current input baud rate. It can have one of the following symbolic values:

`B0`	hangup
`B50`	50 baud
`B75`	75 baud
`B110`	110 baud
`B134`	134 baud
`B150`	150 baud
`B200`	200 baud
`B300`	300 baud
`B600`	600 baud
`B1050`	1050 baud
`B1200`	1200 baud

B1800	1800 baud
B2000	2000 baud
B2400	2400 baud
B4800	4800 baud
B7200	7200 baud
B9600	9600 baud
B19200	19200 baud
B38400	38400 baud
EXTA	external A
EXTB	external B

`sg_ospeed` Contains the current output baud rate. It can have the same values as `sg_ispeed`.

`sg_erase` Contains one of the erase characters. It is only active in cooked mode. For more information refer to the `erase character` section in the description of cooked mode.

`sg_2erase` Contains the second erase character. It is only active in cooked mode. For more information refer to the `erase character` section in the description of cooked mode.

`sg_kill` Contains the character which erases an entire line. It is only active in cooked mode. See the `erase line` section in the description of Cooked Mode for more information.

`sg_flags` Contains various driver control flags and attributes. See the section on Driver Operation Modification Flags for more information.

struct tchars

`t_intrc` Contains the character which causes a `SIGINT` signal to be sent. For more information, see the description of `SIGINT` under the Signal Generation section.

`t_quitc` Contains the character which causes a `SIGQUIT` signal to be sent. See the description of `SIGQUIT` under the Signal Generation section.

`t_startc`	Contains the flow control character which starts transmission after it has been stopped by `t_stopc`. For more information, refer to the section on Flow Control.
`t_stopc`	Contains the flow control character which stops output. For more information see the section on Flow Control.
`t_eofc`	Contains the end-of-file character. Makes the current cooked input line available and causes the next subsequent `read` to return 0 (`EOF`).
`t_brkc`	Contains the input delimiter character. See the Delimit Line section in the description of Cooked Mode.

struct ltchars

`t_suspc`	Contains the character which causes a `SIGTSTP` signal to be sent. For more information, see the description of `SIGTSTP` under the section on Signal Generation.
`t_dsuspc`	Contains the character which causes a delayed transmission of the `SIGTSTP` signal.
`t_rprntc`	Contains the character which causes the current input line to be reprinted. This function only operates in cooked mode. For more information, see the reprint line section under the description of cooked mode.
`t_flushc`	Contains the character which causes an output queue flush. For more information, consult the description of Output Flushing under the description of Output Processing.
`t_werasc`	Contains the character which erases individual words from a line of input. This function only operates in `cooked mode`. For more information, refer to the Erase Word Section under the description of Cooked Mode.
`t_lnextc`	Contains the character which causes the driver to interpret the next character typed as a non-special character. It is usually used when it is desired to input a character which has a special meaning to the `tty` driver. This character is only active in `cooked` and `cbreak` modes.

Alternate Operational Parameters

The `tty` driver supports a System V-compatible interface in which the operational parameters are accessed or modified using the `termio` structure. The structure is defined in `<termio.h>` as follows:

```
struct termio {
    short c_iflag;      /* input modes */
    short c_oflag;      /* output modes */
    short c_cflag;      /* control modes */
    short c_lflag;      /* line discipline modes */
    char c_line;        /* line discipline */
    char c_cc[NCC];     /* control characters */
};
```

Values for the fields in the `termio` structure are defined in `<termio.h>`.

ioctl Functions

The `ioctl` system call is used to change the various driver parameters. `ioctl` takes three arguments: a file descriptor referencing a device, a function code, and a pointer to data whose form depends on the particular function. A list of the functions available with this driver follows.

`TIOCGETP` Get the current `sgttyb` parameters of the `tty` driver. The third argument should be a pointer to an `sgttyb` structure to hold the current settings.

`TIOCSETP` Set the current `sgttyb` parameters of the `tty` driver. The third argument should be a pointer to an `sgttyb` structure containing the desired settings. This request also causes the driver to flush its input buffer if a mode change results.

`TIOCGETC` Get the current `tchars` parameters of the `tty` driver. The third argument should be a pointer to a `tchars` structure to hold the current settings.

`TIOCSETC` Set the `tchars` parameters of the `tty` driver. The third argument should be a pointer to a `tchars` structure containing the new settings for the `tty` driver.

`TIOCGLTC`	Get the current `ltchars` parameters of the `tty` driver. The third argument should be a pointer to an `ltchars` structure to hold the current settings.
`TIOCSLTC`	Set the `ltchars` parameters of the `tty` driver. The third argument should be a pointer to an `ltchars` structure containing the new settings for the `tty` driver.
`TIOCSETN`	This request is identical to `TIOCSETP` except that the input buffer is not flushed if a mode change occurs.
`TIOCEXCL`	Sets "exclusive use" mode. No further `open` requests to the port will be allowed until the current file descriptor is closed.
`TIOCNXCL`	Disables "exclusive use" mode.
`TIOCHPCL`	Causes the `tty` driver to hang up the terminal when the file is closed for the last time. This is useful when the `tty` driver has a line associated with an ACU being used to place outgoing calls.
`TIOCFLUSH`	Flushes all characters waiting in both input and output queues.
`TIOCSTI`	"Stuffs" a character into the input queue of the port. The character is then interpreted as if it had been typed at the terminal. The third argument should point to the character.
`TIOCSBRK`	Sets a hardware break condition on the outgoing line.
`TIOCCBRK`	Clears the break condition on the outgoing line.
`TIOCGPRGP`	Return the current `tty` process group in the `int` pointed to by the third argument.
`TIOCSPGRP`	Sets the `tty` process group to the value in the `int` pointed to by the third argument.
`FIONREAD`	Places the number of currently-readable characters in the input queue in the `long` pointed to by the third argument.
`TIOCLBIS`	The bits set in the `long` pointed to by the third argument are set in the `local mode word`.

TIOCLBIC	The bits set in the `long` pointed to by the third argument are cleared in the `local mode word`.
TIOCLSET	Set the `local mode word` to the value of the `long` pointed to by the argument.
TIOCLGET	Return the value of the `local mode word` in the `long` pointed to by the argument.
TIOASYNON	Enable transmission of `SIGIO` whenever input is available.
TIOASYNOFF	Disable transmission of `SIGIO` whenever input is available.
TCGETA	Get the current parameters of the `tty` driver. The third argument should be a pointer to a `termio` structure to hold the current settings.
TCSETA	Set the current parameters of the `tty` driver. The third argument should be a pointer to a `termio` structure containing the desired settings.
TCSETAW	Same as `TCSETA`, except wait for the output to drain before making the change.
TCSETAF	Same as `TCSETA`, except wait for the output to drain, then flush the input queue, before making the change.

The Control Terminal

The `init` program establishes via the `chcdev` system call a particular terminal as the `control device` for each child it starts. The control device thus established is whatever terminal the child process (usually `login` or `dlsh`) uses for standard input, output, and error channels.

To provide the process with easy access to its control terminal, the special file `/dev/tty` functions as a synonym for the control terminal. When `/dev/tty` is referenced in `open`, the operating system opens the terminal actually established as the control device. Thus if a process must have access to an interactive terminal regardless of I/O redirection performed when it was invoked, it can use `/dev/tty`.

The tty Process Group

To aid command interpreters like `dlsh` in arbitrating between different processes on a single terminal, the `tty` driver allows related process to be placed into a single process group and then associates this group with the terminal. All signals generated by the `tty` driver are then sent to the processes belonging to this group.

The process group of a terminal can be set using the `TIOCSPGRP ioctl` call. This is performed as follows:

```
int pgrp;
pgrp = something;
ioctl(fd, TIOCSPGRP,&pgrp)
```

The `TIOCGPGRP` request returns the current process group for the port:

```
int pgrp;
ioctl(fd, TIOCGPGRP, &pgrp)
```

FILES

`/dev/tty`	Synonym for the control terminal for any process

SEE ALSO

Utility Programs -	`stty, tty, getty, init, dlsh`
System Calls -	`ioctl(), chcdev(), fcntl(), select()`

NAME

`udp` – Internet User Datagram Protocol

SYNOPSIS

In the socket environment:

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

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

In the TLI environment:

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

t = t_open("dev/udp", O_RDWR, info);
```

DESCRIPTION

`UDP` is a simple, unreliable datagram protocol which 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 may 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).

In the TLI environment, the `/dev/udp` device is used to implement the `T_CLTS` type of transport. The `/dev/udp` device is a clonable STREAMS device. The number of minor devices for this driver can be specified by the `SMD_NUDP` parameter in the `/sys/devices/stsocket_info.c` file. The `UDP` transport endpoints are connectonless and are used with the `t_sndudata(3)` and `t_rcvudata(3)` calls of the TLI library.

`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 (i.e., a `UDP` port may not be connected to a `TCP` port). In addition broadcast packets may be sent (assuming the underlying network supports this) by using a reserved broadcast address; this address is network interface dependent.

Options at the `IP` transport level may be used with `UDP`; see `ip(4)`.

DIAGNOSTICS

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

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

SEE ALSO

System Calls -	`getsockopt(), recv(), send(), socket()`
Library Functions -	`t_open(), t_bind(), t_sndudata(), t_rcvudata()`
Device Facilities -	`intro, inet, ip`

NAME

unfs - NFS client I/O driver

DESCRIPTION

The unfs device driver is used to queue up NFS (Network File System) protocol requests from user tasks in kernel mode. These queued up requests are read by the unfsio daemon, encapsulated in RPC (Remote Procedure Call) messages using XDR (External Data Representation) protocol, and sent to the NFS server machine using a UDP (User Datagram Protocol) socket. When a response is received for the NFS request, the response is “written” to the unfs device. The unfs device driver passes this response to the waiting user task in the kernel.

Multiple unfsio daemons can be started to process multiple NFS requests in parallel. The /sys/devices/nfsinfo.c file contains the following structure which can be used to configure parameters for NFS Client operation:

```
struct unfs_info {
    int nnfsnodes; /* num of NFS file system nodes */
    int nnfsmounts;/* num of nfs mount table entries */
    int nopen;     /* Max num of unfsio tasks that
                      can use the unfs device in parallel
    int nports;    /* Max num of NFS requests that
                      can be queued in the kernel */
    int rdsize;    /* Max num of bytes in a read request
    int wrsize;    /* Max num of bytes in a write request
};
```

The rdsize and wrsize should be less than 8K bytes because it is limited by the maximum size of the datagram allowed by the UDP protocol implementation. The rdsize and wrsize values should be chosen properly to match the size of the buffers on the ethernet adapter boards on the “client” and “server” machines.

FILES

/sys/devices/nfsinfo.c

`/dev/unfs`

SEE ALSO

Utility Programs - `unfsio`

NAME

`uvmedrvr` – driver for the Universe™ PCI/VME bridge (PowerPC)

DESCRIPTION

The `uvmedrvr` device driver does the necessary programming to the Universe chip to allow VME access from LynxOS on a VMEbus-based CPU board. The following aspects of this driver are discussed in the sections below:

- VMEbus address space
- VMEbus interrupt handling
- Slave image programming
- Direct Memory Access (DMA)
- Ioctl Commands
- VMEbus error handling

VMEbus Address Space

To access the VMEbus address space under LynxOS, the device driver `uvmedrvr` must be installed. This driver programs the Universe device to map 256 MB of A32/D32 address space at physical address `0xd0000000` and 256 MB of VME short I/O address space at `0xe0000000`. These addresses can be used with the `smem_create()` call to access the VMEbus from an user application.

The kernel virtual address for the A32/D32 address space is `0xd0000000` and `0xc0000000` for the VME short I/O space. These addresses are used to access the VMEbus from a device driver. You can use the functions `recoset()/noreco()` to probe the VMEbus for a VME device.

VMEbus Interrupt Handling

On the Genesis-2 (MVME2600), the RavenMPIC IRQ IRQ5 is assigned for the Universe interrupts. All seven VME interrupts, the `VOWN` interrupt, the VMEbus and PCI Bus Error interrupts and the DMA interrupt are enabled. Any interrupt can be enabled/disabled by modifying the field `lint_en` in the `uvmeinfo` structure defined in `uvmeinfo.c`. Please refer to the Universe User Manual for details on the `lint_en` register.

The VME interrupts from external VME devices must choose from vectors `0x40` to `0xff`. This vector is passed to the `iointset()` call to set up the interrupt handler. For DMA interrupt handler, the vector `DMA_VECT` (`0x3f`) is used and for VMEbus error handler, the vector `VECT_VERR` (`0x3e`) is used.

Slave Image Processing

Slave image programming provides accesses to A32, A24 and A16 spaces. The mapping is as follows:

Space	Physical	Virtual
A32–D32	d0000000-dfffffff	d0000000-dfffffff
A24–D16	e0000000-e0ffffff	c0000000-c0ffffff
A32–D16	e1000000-ef7fffff	c1000000-cf7fffff
A16–D16	efff0000-efffffff	cfff0000-cfffffff

The info file, `uvmeinfo.c` contains user-configurable parameters. The structure of `uvmeinfo` is:

```
struct uvmeinfo {
  int slot;
  unsigned long uvmeregs_addr;/* universe registers addr */
  unsigned char int_line;
  struct pci_slv_img pci_slv0;    /* pci slave images */
  struct pci_slv_img pci_slv1;
  struct pci_slv_img pci_slv2;
  struct pci_slv_img pci_slv3;
  struct vme_slv_img vme_slv0;    /* vme slave images */
  struct vme_slv_img vme_slv1;
  struct vme_slv_img vme_slv2;
  struct vme_slv_img vme_slv3;
  struct vrai_img vrai_img;
  unsigned long statid;       /* Universe's status-id */
  unsigned long  misc_ctl_syscon;
                /* 1 if sys controller, 0 otherwise */
  unsigned long nosysfail;
                /* if sysfail status to be cleared  */
```

```
    unsigned long lint_en;    /* pci intr enable bits */
    unsigned long vint_en;    /* vme intr enable bits */
};
```

Most are set with proper values and it may not be necessary to change. Some important ones are detailed below:

`slot` Dynamically configured.

`uvmeregs_addr` Dynamically configured.

`int_line` Dynamically configured.

`vme_slv_img` The parameters for the VME-slave images. Only image0 is set with some example values for A32 mode access. Other three images are left initialized to 0.

`misc_ctl_syscon`
The flag that permits the system controller to be software-controllable.

`nosysfail` The system comes up with the `SYSFAIL` set. The PCI interrupt status register will show this bit to be set. It can be cleared by setting this flag to 1 (default value is 0).

`lint_en`
`vint_en`
The PCI/VME interrupts can be enabled/masked using these fields.

The driver maps the universe registers at the first available location in `pci-memory/pci_io` space. Any previous mapping done should be cancelled out in the OpenFirmware environment.

Direct Memory Access (DMA)

Two kernel functions `uvme_dma_program()` and `uvme_dma_go()` are provided to cause DMA transfer to take place.

```
uvme_dma_program(struct dma_flags *dmaflags):
```

This function programs the DMA registers to prepare for DMA transfer. The structure `dmaflags` is defined in the include file `<uvme.h>`, the header file that contains the necessary DMA parameters:

```
struct dma_flags {
    unsigned long pci_addr;    /* pci address */
    unsigned long vme_addr;    /* vme address */
    unsigned long byte_cnt;    /* byte count  */
    int direction;             /* VME2PCI or PCI2VME */
    int vdw;                   /* vmebus max datawidth */
    int vas;                   /* vmebus address space */
    int am_pgm;                /* AM code pgm/data */
    int am_sup;                /* AM code sup/np */
    int vct;                   /* Vmebus cycle type */
    int ld64en;    /* enable 64bit pci transactions */
};
```

`uvme_dma_go()` sets the `GO` bit, actually starting the DMA transfer.

Example Driver

The following code fragment from a sample driver illustrates the use of the DMA functions.

```
/* includes for application */
...
...
#include <uvme.h>
#include "/sys/drivers/uvme/universe.h"
/* driver static structure */
static struct st_softc {
    struct universe_regs *uregs;    /* UCSR structure */
    ...
    ...
}
extern ulong uvmeregs_addr; /* an exportable global that
                             * points to the beginning
                             * of the mapped UCSR
                             */
extern int uvme_dma_program();/* programs dma registers */
```

```
extern void uvme_dma_go();/* starts dma transfer */
extern int dma_sem;
void vtestintr(struct v_softc *s);  /* test ISR */

/* install routine */
vtest_install(info)
struct vtestinfo *info;
           /* info structure in /sys/devices directory */
{
    int ioint_id;

   st_softc->uregs=(struct universe_regs *)uvmeregs_addr;

    /* now Universe Registers can be accessed at
     * st_softc->uregs.See /sys/drivers/uvme/universe.h
     * for register structure definition. */

    /* set the interrupt service routine vtestintr with
     * the required vector */

     ioint_id = iointset(info->vector, vtestintr, v_softc);
     return (OK);

}

/* test interrupt service routine */
void vtestintr(struct vtest_softc *s)
{
    /* ISR */
}

/* process DMA */
void do_dma()
{
    struct dma_flags *dmaflags;
    /* fill-in the dma_flags structure */
    dmaflags->pci_addr = 0x81000000;
    dmaflags->vme_addr = 0x09000000;
    dmaflags->byte_cnt = 0x100;
    dmaflags->direction = PCI2VME;
    dmaflags->vdw = VDW_32;
    dmaflags->vas = VAS_A32;
```

```
        dmaflags->am_pgm = AM_DATA;
        dmaflags->am_sup = AM_NP;
        dmaflags->vct = VCT_SC;

        uvme_dma_pgm(dmaflags);
        uvme_dma_go(dmaflags);
        swait(&dma_sem);
    }

        ...

        ...
```

Ioctl Commands

DMA commands

DMA_PGM Program DMA (refer to the structure dmaflags in <uvme.h>).

DMA_GO Start the DMA transaction.

DMA_STOP Stop the DMA transaction.

DMA_HALT Halt the DMA transaction.

Others

Register display commands (refer to <uvme.h> for related structure definitions):

PRINT_CONF_REGS
Print configuration registers

PRINT_INT_CTL_REGS
Print interrupt and control registers

PRINT_DMA_REGS
Print DMA-related registers

PRINT_IMAGE_REGS
Print the slave image registers

PRINT_ERR_REGS
Print error registers.

`REGISTER_POSTED_WRITE`
Register a posted write transaction.

VMEbus Error Handling

Read accesses to the VME space are always coupled; a VMEbus-error received by the Universe results in a target Abort to the PCI Master. This in turn raises a Machine Check Exception. Write accesses can be either posted or coupled. The driver sets this to be posted. This can be altered by toggling the `PWEN` bit in the respective PCI slave image. (Refer to the Universe User Manual for details.) If the writes are posted, Universe doesn't generate a machine check. Instead, a VMEbus Error interrupt is generated, which is handled by the Interrupt Service Routine `verrint()` in the driver.

Due to the nature of asynchronous write, to insure that the VMEbus Error service routine sends the kill signal to the correct thread, a special ioctl command, `REGISTER_POSTED_WRITE` is provided, which can be called from the user code to register to the driver that a posted write is on its way, which makes sure that the `SIGBUS` signal is sent to this process in case there is a bus error.

If this `ioctl` is not called, the driver doesn't send a `SIGBUS` to the process, but prints a message on the console telling that there was a VMEbus error.

The code may look as follows:

```
#include <sys/ioctl.h>
#include <signal.h>


sig_handler()
{
}


main()
{
    signal(SIGBUS, sig_handler);

    fd = open("/dev/uvme0", "O_RDWR");
    ioctl(fd, REGISTER_POSTED_WRITE);

    DoPostedWrite();

    ...
    ...
}


DoPostedWrite()
{
     ...
}
```

SEE ALSO

Device Facilities - `vmedrvr`

NAME

`vmedrvr` – driver for the VMEChip2/VME2PCI bridge (PowerPC)

DESCRIPTION

`vmedrvr` does the necessary programming to the VMEChip2 and VME2PCI chips to allow VME access from LynxOS on a VMEbus-based CPU board. The following aspects of this driver are discussed in the sections below:

- VMEbus address space
- VMEbus interrupt handling
- Slave image programming

VMEbus Address Space

To access the VMEbus address space under LynxOS, the `vmedrvr` driver must be installed. The `vmedrvr` driver programs the VME2PCI device to map 256 MB of A32/D32 address space at physical address `0xd0000000` and 256 MB of VME short I/O address space at `0xe0000000`. These addresses can be used with the `smem_create()` call to access the VMEbus from an user application.

The kernel virtual address for the A32/D32 address space is `0xd0000000` and for the VME short I/O space is `0xc0000000`. These addresses are used to access the VMEbus from a device driver. You can use the functions `recoset()/noreco()` to probe the VMEbus for a VME device.

VMEbus interrupt Handling

LynxOS supports all of the VME interrupts from the VMEchip2 with the exception of the eight software interrupts. The following are the interrupt vectors for the various local bus interrupts generated by the VMEchip2:

GCSR LM0	0x30
GCSR LM1	0x31
GCSR SIG0	0x32
GCSR SIG1	0x33
GCSR SIG2	0x34

GCSR SIG3	0x35
DMAC	0x36
VME bus Int. ack	0x37
Tick timer 1	0x38
Tick timer 2	0x39
VME bus IRQ1	0x3A
Write Post error	0x3C
VME bus SYSFAIL	0x3D
VME bus ACFAIL	0x3F

These interrupts can be enabled by setting the appropriate bits in the VMEchip2. To set the appropriate bits in the VMEchip2, modify the following line in the VME driver:

```
vmechip2->lbint_en = 0x0000007f;
```

The value 0x7f enables all seven external VME bus interrupts. The VME interrupts from external VME devices must choose from vectors 0x40 to 0xFF. This vector is passed to the `iointset()` call to setup the interrupt handler. Please refer to the *MVME 1603/1604 Single Board Computer Programmer's Reference Guide* for more information about VME bus interrupt vectors.

Slave Image Processing

Slave image programming provides accesses to A32, A24 and A16 spaces. The mapping is as follows:

Space	Physical	Virtual
A32–D32	`d0000000-dfffffff`	`d0000000-dfffffff`
A24–D16	`e0000000-e0ffffff`	`c0000000-c0ffffff`
A32–D16	`e1000000-ef7fffff`	`c1000000-cf7fffff`
A16–D16	`efff0000-efffffff`	`cfff0000-cfffffff`

The file vmeinfo.c contains user-configurable parameters. The structure of vmeinfo.c is shown below.

```
struct vmeinfo {
    unsigned long vme2pci_addr;
    unsigned short iomap_baseaddr;
    unsigned short slave_startaddr1;
    unsigned short slave_endaddr1;
    unsigned short addr_offset1;
    unsigned short slave_startaddr2;
    unsigned short slave_endaddr2;
    unsigned short addr_offset2;
    unsigned char int_line;
    int vbr_base;
    unsigned long vmechip2_addr;
};
```

Most are set with proper values and it may not be necessary to change. Some important ones are detailed below:

vme2pci_addr	configuration space base address; dynamically configured
slave_startaddr1	PCI slave start address register 1.
slave_endaddr1	PCI slave end address register 1.
slave_startaddr2	PCI slave start address register 2.
slave_endaddr2	PCI slave start address register 2.
addr_offset2	PCI translation offset
int_line	Interrupt line
vbr_base	vector base to program in VBR of VMEchip2 for interrupts
vmechip2_addr	base address of VMEchip2 registers; dynamically configured.

For more details on the above parameters, refer to the *MVME 1603/1604 Single Board Computer Programmer's Reference Guide*.

SEE ALSO

Device Facilities - `uvmedrvr`

NAME

`wd` – interface to Ethernet (x86)

DESCRIPTION

The `wd` device driver is a special character device to perform communication over Ethernet/IEEE802.3 data link. This driver supports the following Ethernet cards from Standard Micro Systems (and Western Digital Corporation).

Western Digital

- WDLAN-EPRE Ethercard PLUS Elite
- WD8003E Ethercard PLUSRE Ethercard PLUS Elite
- WD8013EBT Ethercard PLUS 16

SMC

- LAN EPRE 8-Bit AUI/BNC for AT bus
- LAN E10TE 8-Bit Twisted pair for AT bus
- LAN EP16E 16-Bit AUI/BNC for AT bus
- 16 Bit cards for AT bus
    - SMC-E16T, SMC-E16, SMC-E16C
    - LANET16, LANEP16E, LANET16C
- 8 Bit cards
    - LANE10TE, LANEPRE

It provides the following `ioctl` functions:

`SIOCSIFADDR`	Used by the `ifconfig` utility to pass-on the Internet address of the Ethernet interface.
`SIOCSIFFLAGS`	Used by the `ifconfig` utility to pass-on the flags to set/reset the interface.

GE_SET_DEV	Sets the interface in raw_ethernet mode. In this mode, the applications can directly communicate to the interface using raw_ethernet sockets.
GE_SET_ADDR	Changes the Ethernet address of the interface.
GE_SET_PROTO	Facilitates the setting up of the protocol range that the raw_ethernet socket needs to receive.
GE_SET_MCAST	Allows the user to set up a list of “multicast addresses” that the interface can receive as destination node using a raw_ethernet socket.
GE_PROM	When programming the interface to receive all packets, this ioctl utility determines whether or not it is meant for this node.
GE_DEBUG_MODE	Puts the interface in debug mode. The data transfers through this interface would be displayed in the debug terminal.
GE_GET_CTRS	Gets the raw_ethernet interface statistics.
GE_GET_SADDR	Gets the current Ethernet address of the interface.
GE_GET_FADDR	Gets the station address as assigned by the manufacturer of the ethernet interface hardware.
GE_CLRIFNAME	Closes the interface for raw_ethernet reception for a raw_ethernet socket.

The /sys/devices/if_wd3einfo.c file has the following structure for configuring the Ethernet interface:

```
struct wd3e_uinfo {
    unsigned long baseio ;
    unsigned long rambase ;
    unsigned long vector ;
#ifndef ps2
    unsigned long enable_16bit;
#endif
};
```

The field `enable_16bit` is applicable only for AT bus 16-bit adapters. Setting this field to 1 sets the card into 16-bit mode. Because of inherent limitations of the AT bus, you can only use 16-bit cards on the 128KB address segment which overlaps the Ethernet card's RAM addresses (e.g., 0xC0000 - 0xDFFFF for the default RAMbase address 0xCC000). By default, this field is set to 0, keeping the Ethernet card in 8-bit mode.

FILES

`/sys/devices/if_wd3einfo.c`

`/dev/wd`

SEE ALSO

Utility Programs -	`ifconfig`
Device Facilities -	`raw_ethernet`
Documents -	See “Raw Ethernet Support” in the LynxOS User’s Guide.

NOTES

Unlike normal devices, this device cannot be operated with open/close `ioctl` calls directly. Instead, the device can be operated through sockets either in `AF_INET` domain or in `AF_RAWETH` domain.

File Formats

NAME

`Intro` – file format general information

DESCRIPTION

The File Format section of the manual contains information about three types of files. The first type includes various configuration and information tables used by LynxOS Utility Programs. Examples of this type include the password file and the terminal information database. The second type are file formats produced by various utilities, such as “tape archives” and “object code” libraries. The final type includes “system” file formats, such as directories and boot blocks.

The **SYNOPSIS** section in the documentation for each file may contain a file name, indicating that the file format described is for some data file which must have the indicated pathname. The **SYNOPSIS** may also include names of C header files, indicating that structure definitions for use with the file format may be found in that header file.

NAME

a.out – executable and object file format files

SYNOPSIS

#include <a.out.h>

DESCRIPTION

The format of the output of the compiler and linker (without the **-k** option) is called a.out format. Executable and object files that are in a.out format begin with a header, which is always the same size.

```
struct exec {
  long a_magic;    /* magic number */
  unsigned long a_text;   /* size of the text segment */
  unsigned long a_data;   /* size of the data segment */
  unsigned long a_bss;    /* size of the bss segment */
  unsigned long a_syms;   /* size of the symbol table */
  unsigned long a_entry;  /* entry point for the program */
  unsigned long a_trsize; /* size of the text relocation*/
  unsigned long a_drsize; /* size of the data relocation*/
};
#define OMAGIC 0407  /* object file */
#define ZMAGIC 0413  /* executable */
```

If the a_magic field is OMAGIC, the file is an object file; if it is ZMAGIC, the file is an executable file.

Following the header are the text (executable code) segment and the data (initialized variables) segments. The text segment and data segment sizes are given by the header fields a_text and a_data.

After the data segment comes relocation information for both the code and data. In a pure executable file, this section is empty; otherwise, a_trsize and a_drsize contain the sizes of the relocation lists. The relocation lists consist of relocation info structures, defined as follows in **a.out.h**:

```
struct relocation_info {
  unsigned long  r_address;  /* Address to relocate */
  unsigned long  r_symbolnum:24,  /* Symbol num in table*/
    r_pcrel:1,  /* PC relative value */
    r_length:2,   /* 0=byte, 1=word, 2=long */
    r_extern:1,  /* Symbol value not included*/
    :4;     /* filler */
};
```

The `r_address` field contains the address in the segment to be relocated. The second `long` value is broken into four subfields, telling how the address is to be adjusted. If `r_extern` is 1, then the value of the symbol given by `r_symbolnum` is added to the value at `r_address`. If `r_extern` is 0, then `r_symbolnum` takes on the same values as the `n_type` symbol table entry field (see below) and thus gives a segment base address to add to the value in `r_address`.

The size of the value to be relocated (byte, word, or long) is given in `r_length`. If `r_address` contains a program counter relative address, `r_pcrel` is set to 1, otherwise 0.

After the relocation lists is the symbol table. The list of symbol table entries contain pointers to symbol names in a string table which immediately follows the symbol table. Each symbol table entry is an `nlist` structure, as follows:

```
struct nlist {
  union {
    char *n_name;   /* in-core pointer to name */
    long n_strx;    /* index into file string table */
  } n_un;
  unsigned char n_type;  /* symbol type --- see below */
  char n_other;    /* filler */
  short n_desc;    /* description for a stab entry */
  unsigned long n_value;  /* value (offset) of symbol */
}
#define N_UNDF 0x00     /* undefined */
#define N_ABS 0x02      /* Absolute */
#define N_TEXT 0x04     /* In text segment */
```

```
#define N_DATA 0x06     /* In data segment */
#define N_BSS 0x08      /* In bss segment */
#define N_COMM 0x12     /* In blank common */
#define N_EXT 0x01      /* or'ed in if external */
#define N_TYPE 0x1e     /* mask for all bits */
#define N_FN 0x1f       /* file name symbol */
#define N_STAB 0xe0     /* mask for stab entry */
```

The `n_strx` field contains the offset from the start of the string table. For use after the file is loaded into memory, `n_strx` is defined in a union with a character pointer, `n_name`. The `n_type` field tells which segment contains the symbol. If any of the high three bits of `n_type` are set, then the symbol table entry is a special “stab” entry, used by a debugger or some other run-time utility. The `n_desc` also pertains to “stab” entries. The field `n_value` contains the value or offset of the symbol from the start of its segment.

NOTE

When the linker is invoked without the **-k** option and without the COFFLD environment variable set, it produces output in `a.out` format. With the **-k** option or the COFFLD environment variable set, it produces output in common object file format (`coff`).

SEE ALSO

Utility Programs - `ld, nm, size`

Library Functions - `nlist()`

File Formats - `stab, coff`

NAME

`boot` – boot block format

SYNOPSIS

#include <disk.h>

DESCRIPTION

Each block device can contain a default bootstrap program in an initialized `boot` block. The `boot` block is always block zero. It contains the address at which the `boot` program is to be loaded, the size of the `boot` program, and pointers to the disk blocks actually containing the code.

```
#define B_DIRECT 10      /* direct pointers in boot block */
#define B_IDIRECT 10     /* indirect pointers in boot block */
struct bootblock {
  long b_startaddr;     /* load address */
  long b_size;          /* size of boot file */
  long b_direct[B_DIRECT];     /* direct block pointers */
  long b_idirect[B_IDIRECT];   /* indirect block pointers */
};
```

The direct block pointers (`b_direct`) are block numbers containing the first `B_DIRECT` blocks (almost certainly ten) in the `boot` file. The indirect block pointers point to `B_IDIRECT` blocks, which are blocks containing 128 data block addresses each. Thus the total file size of the bootstrap program must be less than or equal to 10 x 512 + 10 x 128 x 512, 660,480 bytes.

The bootblock is normally created with the `makeboot` program.

SEE ALSO

Utility Programs - `makeboot, reboot`

System Calls - `reboot()`

NAME

`bplane.conf` – format of the configuration file for a SCMP network

SYNOPSIS

`/etc/bplane.conf`

DESCRIPTION

The file `/etc/bplane.conf` contains the configuration entries corresponding to each MPU module connected over the VMEbus via SCMP. This file is read by the `bpconfig` utility that passes the configuration to the SCMP driver. When adding or deleting a MPU module to or from the SCMP network `/etc/bplane.conf` must be changed.

Each MPU module sharing the VMEbus must have an exclusive entry in the `bplane.conf` file. An entry consists of a single line containing the following fields:

Type Boot-Flag DRAM-Address IO-Address IP-Address Name

The fields must be separated by one or more space or tab characters.

Type — For 68k boards, this field must be set to 1. For PowerPC boards, this field must be set to 2.

Boot-Flag — This field is currently unused. This field must contain zero.

DRAM-Address — Each MPU module's on-board DRAM memory must be configured to make them accessible from other modules at an exclusive VME base-address. This address is a multiple of the on-board DRAM size. Be careful that the addresses between any two modules do not overlap by taking into account the on-board DRAM available in each module.

IO-Address — A set of on-board registers in each MPU module should be available for other modules to do intermodule communication. The base address starts at 0xFFFF0000 for MVME 68K CPU boards and

0xCFFF0000 for MVME 1600 series boards. Each module has 16 addresses starting with the offset given in this field.

IP-Address This field contains the IP address for each MPU module sharing the VME bus. An entry for this field must be either given in standard IP dotnotation (4 dot-separated decimal numbers) or a symbolic name available in the `/etc/hosts` file. The resulting Internet address should belong to an exclusive network.

A # (pound-sign) anywhere in the file indicates a comment that extends to the end of the line.

EXAMPLE

The following lines show one possible configuration of three MPU modules within a MVME 68K SCMP system. Because it is the first entry listed, the module corresponding to `scmp-cpu1` must be set up as the SCMP master (also known as the system controller).

```
#Type Boot-Flag DRAM-Address IO-Address Hostname

1     0         0x4000000    0xFFFFCC00 scmp-cpu1
1     0         0x8000000    0xFFFFCC10 192.1.1.2
1     0         0xA000000    0xFFFFCC20 192.1.1.3
```

FILES

`/etc/hosts`

`/etc/bplane.conf`

`/etc/config/netconfig.bplane`

SEE ALSO

Utility Programs – `bpconfig, slaveboot`

NAME

`cfg` – device driver configuration file format

DESCRIPTION

Device drivers configured into LynxOS are defined by their `cfg` file, which resides in the conf directory. The `cfg` file is referenced by `kernconfig` as part of the kernel configuration process. When a line of the form

`drivers` *drv1*, *drv2*, `...`

is encountered by `kernconfig`, it processes configurations files

drv1`.cfg`, *drv2*`.cfg ...`

A `cfg` file is a table of driver characteristics which describes one or more device drivers. This table describes device drivers by giving entry point names, and lists which devices will be installed for each driver. Each entry in the table is a line of the form

type : *field_list*

where *type* is a single character which indicates the entry type, and *field_list* is a colon-separated list of values appropriate to the particular entry type. Entries may be split across multiple lines by ending a line with a backslash (“\”) and resuming the entry on the next line. The end of the last field on a line in a multi-line entry should be marked with a colon, as should the start of the first field on a continuation line:

`C:a` *driver* : *field* : *field* : *field* : *field* : \
:*field*:*field*

No spaces may follow the backslash on a line which is to be continued.

The entry types of the configuration table are as follows:

`C` Character/STREAMS Device Driver Description:
`C:`*name*`:open:close:read:write:select:control:\`
`install:uninstall:strtab`

The *name* field may be any string which describes the driver.

For character device drivers the rest of the fields are the C entry points except for the last field which may be ignored. Any entry point may be omitted, in which case it is assumed to be `ionull`. Care should be taken to ensure that the names are listed in the correct order.

For STREAMS device drivers, the last field should have the name of the `streamtab` structure for that driver and the other fields may be ignored.

`B` Block Device Driver Entry.

Format is the same as character driver entries, except of course that they start “B” with instead of “C”.

`D` Major Device Entry:

`D`:*name*:*info_block*:*raw_partner*

name may be any string which describes the major device.

info_block is the name of a C data structure, the address of which will be passed to the driver upon device installation at boot time. Generally, the *info_block* is a structure defined in a module in the device library (`/sys/devlib.a`) containing interrupt addresses, port numbers, and other relevant items for use by the driver. The field may be omitted, in which case a null will be passed to the driver upon installation.

raw_partner
is only meaningful for block devices. It should be set to the name of the character major device which provides a `raw` interface to the same physical device as the block entry. This field must be present if there is a `raw` device; otherwise, it may be omitted.

For STREAMS device drivers, the *info_block* and *raw_partner* fields may be omitted.

`N` Node Table Entry:

`N:`*name*`:`*minor_number*`:`*perm*

name is the file name for this node.

minor_number
is combined with the major number of the most-recently declared device to form the node device number (see below). An entry will be written to the nodetab file to create the actual node later with `mknod`.

perm is the permissions with which the node should be created. *perm* is an octal number. The default is 0666. Only the *perm* field is optional.

`L` Clonable Device Driver Entry:

`L:`*name*`:`*perm*`:`

name is used as the file name for this node. The major device number corresponding to the position of this entry is assigned as the minor device number for the file name to be created. The major device number for this file name is that of the `clone` STREAMS device driver. An entry will be written to the `nodetab` file with these major and minor device numbers to create the actual node later with `mknod`.

perm is the permissions with which the node should be created. *perm* is an octal number. This field is optional and its default value is 0666.

`M` STREAMS Module Entry:

`M:`*name*`:`*streamtab*`:`

name is used to identify the STREAMS module in the STREAMS related `ioctl` commands.

streamtab is the name of the *streamtab* structure defined for that module. This information is processed by the `config` utility to create an entry of structure `fmodsw` for each module in an array called `fmod_sw` in the configuration header file.

`S` Special Device Entry:

`S`:*name*:*minor_number*

name is a C constant name (for a `#define`), which will be included in a special include file.

minor_number
The name will be defined to the device number formed from *minor_number* and the current major number. The primary use of this entry is to define the system console device, which is almost always the first `tty` defined.

`n` Raw `nodetab` Entry:

`n`:*nodetab_line*

nodetab_line is dumped directly to the nodetab output of config.

This entry type is most often used to establish symbolic links to other devices. In that case, an "`n`" entry has the form:

`n`:*new_name*:`@`*real_device*`::`
where

new_name is the name of the link

real_device is the target of the link.

`I` Included `sub-config-file`:

`I`:*file*
file is read and interpreted as if it were actually part of the parent file. Include files may be nested to a certain extent.

The order of the entries in the file dictate which devices correspond to which drivers, and which nodes correspond to which devices. The first entry in the configuration file should be a driver entry. After the driver should come the first major device, then each of the minor devices for that major device. More major device and node lists may follow. The sequence then repeats for the next driver. Thus, the configuration file takes the form:

```
device driver entry
device entry
node entry
node entry
...
device entry
node entry
node entry
...
device driver entry
device entry
node entry
node entry
...
```

and so on. `Raw nodetab` entries and STREAMS module entries may be placed anywhere, though they are usually put at the end of the file. Blank lines and lines beginning with “`#`” are ignored.

The `config` program operates by reading in the entries and building lists of drivers and related devices. Major device numbers are assigned positionally; the first device for a driver is numbered 0, the next 1, and so on. Device types (character or block) for nodes are implied by driver association. When the file has been completely read, `config` creates the main configuration header file to be compiled with `conf.c` and linked with LynxOS, another header file containing any special device definitions, and the `nodetab` to create all the nodes.

SEE ALSO

Utility Programs – `kernconfig`

File Formats – `kernconfig`

NAME

`coff` – common object file format

SYNOPSIS

#include <coff.h>

DESCRIPTION

When a program is linked using the `-k` option, or with the `COFFLD` environment variable set, the linker will produce an output file in the COFF format (Common Object File Format).

A `coff` file is made up of a file header, an optional header, some number of section headers, data sections, relocation information, line number information, a symbol table, and a string table.

```
file header
optional header
section 1 header
...
section n header
data for section 1
...
data for section n
relocation information for section 1
...
relocation information for section n
line number information for section 1
...
line number information for section n
symbol table
string table
```

The relocation information, line number information, symbol table, and string table may be missing if they were removed by the `strip` utility. After linking, there will be no relocation information unless `ld` was invoked with the `-r` option. If the symbol table does not contain any symbols with names longer than eight characters, the string table will be empty.

File Header

The format of the file header is:

```
struct filehdr {
  unsigned short       f_magic;  /* magic number*/
  unsigned short       f_nscns;  /* number of sections*/
  long                 f_timdat; /* time and date stamp*/
  long                 f_symptr; /* file ptr to symbol table */
  long                 f_nsyms;  /* no. of symbols in table */
  unsigned short       f_opthdr; /* size of optional header*/
  unsigned short       f_flags;  /* flags*/
};
#define MCOFFMAGIC 0520          /* Motorola COFF Magic number:
                                    Used  for Sys V objects and
                                    executables on M68K and for
                                    Lynx objects on  M68K */
#define ZCOFFMAGIC 0415          /* Lynx COFF Magic number: Used
                                    for Lynx executables on I386
                                    and M68K */
#define COFFMAGIC  0514          /* Interactive COFF Magic
                                    number: Used for Sys V
                                    objects and executables on
                                    I386 and for Lynx objects on
                                    I386 */
```

Three `coff` magic numbers are supported: `MCOFFMAGIC`, `ZCOFFMAGIC`, and `COFFMAGIC`. If the `f_magic` field is `MCOFFMAGIC`, the file is either a Motorola 68K System V object file or executable, or a Lynx object file for the Motorola 68K platform. If the `f_magic` field is `ZCOFFMAGIC`, the file is a Lynx executable for either the Intel 80x86 or Motorola 68K platform. If the `f_magic` field is `COFFMAGIC`, the file is either an Interactive System V object file or executable, or a Lynx object file for the Intel 80x86 platform.

Optional Header

The format of the optional header is:

```
struct aouthdr {
  short magic;        /* magic number */
  short vstamp;       /* version stamp */
  long tsize;         /* text size in bytes, padded */
  long dsize;         /* initialized data size */
```

```
  long bsize;         /* uninitialized data size */
  long entry;         /* entry point */
  long text_start;    /* base of text for this file */
  long data_start;    /* base of data for this file */
};
#define OMAGIC    0407  /* Object file  */
#define ZMAGIC     0413  /* Executable file  */
```

Two `a.out` magic numbers are supported: `OMAGIC` and `ZMAGIC`. If the magic field is `OMAGIC`, the file is an object file; if it is `ZMAGIC`, the file is an executable file.

Section Header

The format of each section header is:

```
struct scnhdr {
  char s_name[8];  /* section name  */
  long s_paddr;    /* physical address  */
  long s_vaddr;    /* virtual address  */
  long s_size;     /* section size  */
  long s_scnptr;   /* file ptr to section raw data*/
  long s_relptr;   /* file ptr to relocation info  */
  long s_lnnoptr;  /* file ptr to line no. info  */
  unsigned short s_nreloc; /* no. of relocation entries  */
  unsigned short s_nlnno;  /* no. of line no. entries  */
  long s_flags;    /* flags  */
};
```

The start of the relocation information is `s_relptr` from the section header. If there is no relocation information, `s_relptr` is 0.

Relocation Information

If a file contains relocation information, each relocation entry will be in the following format:

```
struct reloc {
  long r_vaddr;  /* virtual address of reference */
  long r_symndx;  /* index into symbol table */
  unsigned short r_type;  /* relocation type */
};
```

If relocation information has not been removed, there will be one `reloc` structure for each re-locatable reference in the text or data of the file.

Line Number Information

If a file contains line number information, each line number entry will be in the following format:

```
struct lineno {
  union {
    long l_symndx;  /* symtbl index of func name  */
    long l_paddr;  /* paddr of line number  */
  } l_addr;
  unsigned short l_lnno;    /* line number  */
};
```

Symbol Table

If a file contains a symbol table, each symbol table entry will be in the following format:

```
#define SYMNMLEN  8
struct syment {
  union {
    char _n_name[SYMNMLEN];    /* symbol name */
    struct {
      long _n_zeroes;    /* 0 if name is in str table */
      long _n_offset;  /* location in string table  */
    } _n_n;
    char *_n_nptr[2];
  } _n;
  unsigned long n_value;      /* value of symbol */
  short n_scnum;      /* section number */
  unsigned short n_type;      /* type */
  char n_sclass;      /* storage class */
  char n_numaux;      /* no. of aux entries */
};
#define n_zeroes _n._n_n._n_zeroes
#define n_offset _n._n_n._n_offset
#define n_nptr   _n._n_nptr[1]
```

Some symbols require more information than a single entry can provide. If this is the case, the symbol table entry will be followed by auxiliary entries, which are the same size as a symbol table entry, and have the following format:

```
#define DIMNUM 4  /* Number of dimensions stored in
    auxiliary entry for array type variables  */
#define FILNMLEN 14 /* File name length in Lynx-14 chars  */
union auxent {
  struct {
    long x_tagndx;
    union {
      struct {
        unsigned short x_lnno;
        unsigned short x_size;
      } x_lnsz;
      long x_fsize;
    } x_misc;
    union {
      struct {
        long x_lnnoptr;
        long x_endndx;
      } x_fcn;    /* Function */
      struct {
        unsigned short x_dimen[DIMNUM];
      } x_ary;      /* Array */
    } x_fcnary;        /* Function - Array */
    unsigned short x_tvndx;
  } x_sym;          /* Symbol table related */
  struct {
    char x_fname[FILNMLEN];
  } x_file;        /* File name */
  struct {
    long     x_scnlen;
    unsigned short    x_nreloc;
    unsigned short    x_nlinno;
  } x_scn;        /* Section */
  struct {
    long     x_tvfill;
    unsigned short    x_tvlen;
    unsigned short    x_tvran[2];
  } x_tv;
};
#define x_name   x_file._x_name
```

```
#define x_nptr   x_file._x_xptr[1]
#define x_zeroes x_file._x_x._x_zeroes
#define x_offset x_file._x_x._x_offset
```

String Table

The string table, if it exists, consists of the length of the string table, in the first four bytes, followed by the null terminated strings which did not fit in the `_n_name` fields of the `syment` structures in the symbol table. Offsets into the string table will begin at 4 due to the string table size stored in the first four bytes.

SEE ALSO

Utility Programs - `ld, strip`

File Formats - `a.out`

NAME

`config` – LynxOS configuration table format

DESCRIPTION

The LynxOS operating system can be configured to include as many or as few device drivers and devices as necessary for an installation. The configuration process revolves around a table, which is read and interpreted by the `config` utility. This table describes device drivers by giving entry point names, and lists which devices will be installed for each driver. Each entry in the table is a line of the form:

```
type:field list
```

where type is a single character which indicates the entry type, and field list is a colon-separated list of values appropriate to the particular entry type. Entries may be split across multiple lines by ending a line with a backslash (“\”) and resuming the entry on the next line. The end of the last field on a line in a multi-line entry should be marked with a colon, as should the start of the first field on a continuation line:

```
C:a driver:field:field: \
    :field:field: \
    :field:field
```

No spaces may follow the backslash on a line which is to be continued.

The entry types of the configuration table are as follows:

`C` Character/STREAMS Device Driver Description

```
C:name:open:close:read:write:\
select:control:install:uninstall:strtab
```

The name field may be any string which describes the driver. For character device drivers the rest of the fields are the C entry points except for the last field which may be ignored. Any entry point may be omitted, in which case it is assumed to be `ionull`. Care should be taken to ensure that the names are listed in the correct order.

For STREAMS device drivers, the last field should have the name of the `streamtab` structure for that driver and the other fields may be ignored.

`B` Block Device Driver Entry

Format is the same as character driver entries, except of course that they start “B”with instead of “C”.

`D` Major Device Entry

`D:name:info-block:raw-partner`

The `name` may be any string which describes the major device. The `info-block` is the name of a C data structure, the address of which will be passed to the driver upon device installation at boot time. Generally, the `info-block` is a structure defined in a module in the device library (`/sys/devlib.a`) containing interrupt addresses, port numbers, and other relevant items for use by the driver. The field may be omitted, in which case a null will be passed to the driver upon installation. The `raw-partner` field is only meaningful for block devices. It should be set to the name of the character major device which provides a `raw` interface to the same physical device as the block entry. This field must be present if there is a `raw` device; otherwise, it may be omitted.

For STREAMS device drivers, the `info_block` and `raw-partner` fields may be omitted.

`N` Node Table Entry

`N:name:minor-number:perm`

The name will be used as the file name for this node. The minor-number will be combined with the major number of the most recently declared device to form the node device number (see below). An entry will be written to the nodetab file to create the actual node later with `mknod`. The perm field is the permissions with which the node should be created. perm is an octal number. The default is 0666. Only the `perm` field is optional.

L — Clonable Device Driver Entry

`L:name:perm:`

The name will be used as the file name for this node. The major device number corresponding to the position of this entry is assigned as the minor device number for the file name to be created. The major device number for this file name is that of the `clone` STREAMS device driver. An entry will be written to the `nodetab` file with these major and minor device numbers to create the actual node later with `mknod`. The `perm` field is the permissions with which the node should be created. `perm` is an octal number. This field is optional and its default value is 0666.

M — STREAMS Module Entry

`M:name:streamtab:`

The name is used to identify the STREAMS module in the STREAMS related `ioctl` commands. The `streamtab` entry is the name of the `streamtab` structure defined for that module. This information is processed by the `config` utility to create an entry of structure `fmodsw` for each module in an array called `fmod_sw` in the configuration header file.

S — Special Device Entry

`S:name:minor-number`

The `name` should be a C constant name (for a `#define`), which will be included in a special include file. The name will be defined to the device number formed from `minor-number` and the current major number. The primary use of this entry is to define the system console device, which is almost always the first `tty` defined.

n — Raw `nodetab` Entry

`n:nodetab-line`

The nodetab-line is dumped directly to the nodetab output of config. This entry type is most often used to establish symbolic links to other devices. In that case, an "`n`" entry has the form:

```
n:new-name:@real-device::
```

where `new-name` is the name of the link, and `real-device` is the target of the link.

`I` Included `sub-config-file`

```
I:file
```

The `file` is read and interpreted as if it were actually part of the parent file. Include files may be nested to a certain extent.

The order of the entries in the file dictate which devices correspond to which drivers, and which nodes correspond to which devices. The first entry in the configuration file should be a driver entry. After the driver should come the first major device, then each of the minor devices for that major device. More major device and node lists may follow. The sequence then repeats for the next driver. Thus the configuration file takes the form:

```
device driver entry
device entry
node entry
node entry
...
device entry
node entry
node entry
...
device driver entry
device entry
node entry
node entry
...
```

and so on. `Raw nodetab` entries and STREAMS module enteries may be placed anywhere, though they are usually put at the end of the file. Blank lines and lines beginning with "`#`" are ignored.

The `config` program operates by reading in the entries and building lists of drivers and related devices. `major` device numbers are assigned positionally; the first device for a driver is numbered 0, the next 1, and so on. Device types (character or block) for nodes are implied by driver association. When the file has been completely read, `config` creates the main configuration header file to be compiled with `conf.c` and linked with LynxOS, another header file containing any special device definitions, and the `nodetab` to create all the nodes.

SEE ALSO

Utility Programs – `config, mknod, devices, drivers`

NAME

`config.snmp` – SNMP master daemon configuration file

DESCRIPTION

This configuration file contains information that controls several aspects of the Simple Network Management Protocol (SNMP) Master agent operation. The Master agent (`snmpd`) reads this file on execution and initializes the various controlling parameters within. The control parameters belong to the following groups:

- initial values for sysContact and sysLocation
- community-based access control
- community-based naming
- trap destinations
- choice of transport protocols
- sub-agent access control
- sub-agent protocol parameters
- relationship of subagents to community-based naming

Each of these groups is discussed in more detail in the following sections.

Initial Values for sysContact and sysLocation

The `INITIAL` directive allows the initial values for the "sysContact" and "sysLocation" variables from MIB-II to be specified. The syntax for an initialization directive is shown below:

```
INITIAL <varName> <String>
<varName> ::= sysLocation | sysContact
```

If the token "string" contains spaces, line breaks, tabs, etc., the "string" must be enclosed in quotes. Values may also be specified in hexadecimal notation. The configuration file processing routines handle the mapping of <CR> (carriage return) and <LF> (line feed) characters into Network Virtual Terminal (NVT) strings.

Note that these directives have an effect only when the agent is started with a non-existent non-volatile memory file. Changing the values in the configuration file and forcing the file to be re-read using a `HUP` signal will not cause the values of the MIB variables to change. This feature helps avoid confusing other managers that may have re-configured those variables.

The following example illustrates the use of the `INITIAL` directive in a configuration file.

```
INITIAL sysLocation "Lynx Real Time Systems.Inc, San Jose, CA
95124"
INITIAL sysContact "Technical Support Email: <support@lynx.com>"
```

Of course, the actual values you choose will be specific to each system. If no `INITIAL` directive is present for a variable, the default value chosen by the agent is the empty string.

Identifying Transport Mappings

The agent may make use of multiple interfaces for SNMP and SMUX traffic. For each interface that will be used to listen for SNMP packets or SMUX connections, there must be one `TRANSPORT` definition in the configuration file. If no transport definitions are present, the defaults are used. If any SMUX transport mappings are present in the configuration file, the default is not assumed. Likewise, if any SNMP transport mappings are present in the configuration file, the default is not assumed.

Each transport mapping describes an interface over which the agent listens for SMUX connections from sub-agents or SNMP requests from managers.

The syntax for the `TRANSPORT` clause is shown below. When appl_protocol is SNMP, transport_protocol defaults to UDP. If appl_protocol is SMUX, transport_protocol defaults to TCP. The IP address in faddr defaults to that of the agent's host. The IP port number defaults to the values in `/etc/services` for SNMP and SMUX protocols.

The syntax for the TRANSPORT clause in the agent configuration file is:

```
TRANSPORT <name> SNMP [OVER UDP SOCKET] [AT <addr>]
TRANSPORT <name> SMUX [OVER TCP SOCKET] [AT <addr>]

addr        ::= <ip-kind> | <rfc1449addr> | <full-ip>
ip-kind     ::= <hostid> | <hostid> <portid> | <portid>

hostid      ::= <hostname> | <ip>
portid      ::= [PORT | : ] <#>

full-ip     ::= <ip>:<#>
ip          ::= <#>.<#>.<#>.<#>

rfc1449addr ::= <ip>/<#>
```

Note that there are many different ways of entering an IP address.

The following example configures an agent to accept SMUX connections at both the standard port and at a non-standard port. It also configures the agent to accept SNMP traffic only at a non-standard port.

```
TRANSPORT extraordinary SNMP OVER UDP SOCKET AT PORT 11161
TRANSPORT ordinary      SMUX OVER TCP SOCKET AT PORT 199
TRANSPORT special       SMUX OVER TCP SOCKET AT PORT 11199
```

The maximum number of concurrent SNMP and SMUX transports is limited by your target system's limits on the number of open sockets or file descriptors per process.

Using Community Strings

Community strings serve many functions in SNMP. Their most common use is to provide a weak form of authentication. In this capacity, they are part of the access control framework for SNMP. They are also used to help name pieces of information in the network. The following example shows how communities can be used:

1. At IP address 192.146.153.65 at port 161, using community string “public”, the value of {sysContact 0} is“Fred”, and only read access is permitted.

2. At IP address 192.146.153.65 at port 161, using community string “RenMinRiBao”, the value of {sysContact 0} is “Fred”, but both read and write access are permitted.
3. At IP address 192.146.153.65 at port 161, using community string “subsystem1”, the value of {sysContact 0} is “Joe”, and only read access is permitted.

In this example, two of the community strings provide access to the same information, but with different access control policies. This third community string provides access to another set of information. Note that the manager requires external knowledge to determine whether the first two refer to the same information or to two different pieces of information that happen to have the same value.

A community string may be thought of as a key into a database table with the following information:

1. Spatial Semantics—which set of information (where a system may have multiple instances of the same information classes)
2. Temporal Semantics—when was the information retrieved or when will changes take effect (there may be caching, changes may require a reboot to take effects etc.)
3. Access Control Policy—who is allowed to use this information and what operations are allowed?

To support all the semantics carried by community strings, there are several configuration file entries.

`COMMUNITY` entries determine which community strings are valid. If no `COMMUNITY` entries are present, *any* community will be accepted by the agent. For each community, the set of supported operations is specified by the ALLOW clause. The MEMBERS clause in a community entry restricts the set of managers allowed to use that community. If the MEMBERS clause is absent from a COMMUNITY entry, any manager may use that community.

```
<communityDefinition> ::= COMMUNITY <communityName>
                             ALLOW <op> [,<op>]* [OPERATIONS]
                             [AS ENTITY <entityName>]
                               (MEMBERS <addr> [,<addr>] ]

<communityName>         ::= <name>

<op>                    ::= ALL | GET | SET | TRAP | RESPONSE |
                           GET-NEXT | GET-BULK | INFORM | NOTIFY
```

The keyword ALL serves as a shorthand for listing the possibilities.

The optional AS ENTITY clause provides any naming semantics associated with this community. An entity definition associates spatial/temporal properties with a label. The spatial/temporal entity is defined using the following grammar:

```
<entityDefinition> ::= ENTITY <EntityName>
                            DESCRIPTION <Strings>
                              [WITH TIME DOMAIN <oid>]

<entityName>       ::= <name>
```

The next example shows how a system might be configured to provide read-only and read-write access both to its standards resources as well as a special subsystem.

```
ENTITY SpecialSubsystem
    DESCRIPTION "The ultra-enhanced subsystem"

COMMUNITY public
    ALLOW GET OPERATIONS

COMMUNITY public1
    ALLOW GET OPERATIONS
    AS ENTITY SpecialSubsystem

COMMUNITY "YingXio'ng"
    ALLOW ALL OPERATIONS

CONMMNITY "Jia-oTo-ng"
    ALLOW ALL OPERATIONS
    AS ENTITY SpecialSubsystem
    MEMBERS bigboss.megalith.com, 192.146.153.65
```

Thus, to access information from the "ultra-enhanced subsystem", a manager would have to use either community "public1" or "Jia-oTo-ng". Using community "public" or "YingXio'ng" would access the default spatial-temporal entity. Furthermore, use of the community "Jia-oTo-ng" is restricted to operations with a source address from system "bigboss.megalith.com" or 192.146.153.65; operations with any other source address would not be accepted with this community.

Controlling Trap Destinations

`MANAGER` entries in the configuration file define SNMP trap destinations as shown below.

```
<trapDestination> ::= MANAGER <addr>
    [ON TRANSPORT <transportName>]
    [SEND [ALL | NO] TRAPS]
    [TO PORT <#> ]
    [WITH COMMUNITY <String>]]
```

See the previous section for a description of the meaning of transportName. If the transport is not specified, the default transport will be the one used for the primary SNMP traffic.

If the `PORT` clause is not supplied, snmp-trap from your systems `/etc/services` will be used.

If the `COMMUNITY` clause is absent, the community string “public” will appear in the traps that are sent.

The following example requests that all traps be sent to manager “fred” at both port 162 and 11162, from the non-standard snmp port defined above, using a community string of “deep dark secret”.

```
MANAGER fred ON TRANSPORT extraordinary
    SEND ALL TRAPS TO PORT 162
    WITH COMMUNITY "deep dark secret"
    SEND ALL TRAPS TO PORT 11162
    WITH COMMUNITY "deep dark secret"
```

The number of trap destinations is limited only by available memory.

Controlling Sub-agent Communications

There are two major aspects of sub-agent communications that may be controlled through the configuration file. The first is sub-agent access control and naming, and the second is sub-agent protocol parameters.

The grammar for the control of subagent access and associating subagents to specific entities is fairly complex, owing to the wide range of possible configurations that might be needed. There are two types of subagent access control statements: those explicitly permitting access, and those denying access.

```
<subagentControl> ::= <allowSpec> | <denySpec>

<allowspec>       ::= ALLOW <subagentId> [ON <hostspec>]
                   WITH <passwordSpec> [<entitySpec>] [<timeout>]

<denySpec>        ::= DENY <subagentId> ON <hostSpec>
                       WITH <passwordSpec>
```

There are a number of support productions required to complete these

subagent access control descriptions.

```
<subagentId>      ::= SMUX SUBAGENT <oid> |
                              UNSPECIFIED SMUX SUBAGENT[S]
<hostSpec>        ::= HOST <hostid> |
                              UNSPECIFIED HOST[S]
<passwordSpec>    ::= PASSWORD <string>
                              UNSPECIFIED PASSWORD[S]
<entitySpec>      ::= AS ENTITY <entityName>
<timeout>         ::= USING <specificTimeout> TIMEOUT
<specificTimeout> ::= <number> SECOND[S] | No
```

The relationship between subagents and spatial/temporal entities is closely connected to the extended community string semantics described above.

The `USING ... TIMEOUT` clause allows the system to be configured to robustly handle the case where a sub-agent becomes non-responsive. If a subagent does not produce a response to a request within the specified timeout, the agent terminates the connection.

Although not enforced by the grammar, the following ordering in the configuration file appears to be the most natural for both reading and writing:

1. specific subagents
   - on specific hosts
     i) with specific passwords
     ii) with unspecified passwords
   - on unspecified hosts
     i) with specific passwords
     ii) with unspecified passwords
2. unspecified subagents
   - on specific hosts
     i) with specific passwords

ii) with unspecified passwords

- on unspecified hosts

  i) with specific passwords

  ii) with unspecified passwords

The following examples should make all of this much clearer. The first example requires that all subagents connect from localhost, and sets a 2-second timeout for all of them.

```
ALLOW UNSPECIFIED SMUX SUBAGENTS ON HOST localhost
              WITH UNSPECIFIED PASSWORDS
               USING 2 SECOND TIMEOUT
DENY UNSPECIFIED SMUX SUBAGENTS ON UNSPECIFIED HOSTS
               WITH UNSPECIFIED PASSWORDS
```

We can elaborate on the previous example by identifying a specific subagent as responsible for the information belonging to a specific spatial/temporal entity. Furthermore, we can impose a different responsiveness requirement on that subagent.

```
ALLOW UNSPECIFIED SMUX SUBAGENTS ON HOST localhost
              WITH UNSPECIFIED PASSWORDS
               USING 2 SECOND TIMEOUT
ALLOW SMUX SUBAGENT 1.3.6.1.2.1 ON HOST localhost
              WITH PASSWORD 0x0011223344556677
              AS ENTITY SpecialSubsystem
               USING 42 SECOND TIMEOUT
DENY UNSPECIFIED SMUX SUBAGENTS ON UNSPECIFIED HOSTS
               WITH UNSPECIFIED PASSWORDS
```

The example above says that when the mibII subagent connects from localhost using a password of Ox0011223344556677, that it will be considered to belong to the spatial and temporal domain of SpecialSubsystem (defined above), and that it will be allowed to take up to 42 seconds to respond to requests before being disconnected.

Reloading the Configuration File

The agent may be told to re-read its configuration file by giving the agent process a `HUP` signal. This allows a system administrator to change access restrictions without bringing down the agent. Note that any error message go to the agent's stderr, which is not necessarily the same as that of the process issuing the HUP signal. This action causes the agent to issue a warm start trap, if traps are enabled in the agent configuration file.

SEE ALSO

Utility Programs - `snmpd`, `mibIIagent`, `lynxos_agent`

NAME

`core` – format of process memory image

SYNOPSIS

`#include <proc.h>`

DESCRIPTION

Whenever a program is terminated by LynxOS because of abnormal conditions like memory violations, illegal instructions, bus errors, and user-generated quit signals, the process memory image is dumped in a file called `core`. This file is always created in the process' current working directory.

The `core` file consists of three areas: user area, per-thread area, and data dump stored in that order. The user area consists of `pssentry` structure as defined in `<proc.h>`. The size of user area is 4096 bytes (one page). The rest of the `core` file consists of per-thread context and stacks, and data dumps.

The per-thread context and stack is composed of two parts: the per-thread context and the thread stack, stored in that order. The structure of the per-thread context is the `core_st_t` structure defined in `<proc.h>`. The structure in the `core` file is not padded; the per-thread stack follows immediately in the file, and it is rounded up to the nearest number of pages. The first thread context in the `core` file will be the context of the thread which was running when the process died. The contexts and stacks of the other threads in the process are dumped after the context of the first thread. The other threads are not dumped in any particular order.

The data dump includes `DATA` section, `BSS` section and dynamically allocated memory for the entire process. The sizes of stack dump and data dump are rounded up to the nearest page boundary. The three fields in `pssentry` `ssize`, `data_len`, and `bss_len` contain the actual sizes of the stack, `DATA` section, and `BSS` section plus dynamically-allocated memory.

The `core` file is used by utilities such as `st` and `gdb` for post-mortem debugging.

SEE ALSO

Utility Programs -	`st, gdb`
System Calls -	`sigvec()`
File Formats -	`a.out`

NAME

crontab – tables for driving cron

DESCRIPTION

A crontab file contains instructions to the cron(1) daemon of the general form: "run this command at this time on this date". Each user has their own crontab, and commands in any given crontab will be executed as the user who owns the crontab. UUCP and News will usually have their own crontabs, eliminating the need for explicitly running su(1) as part of a cron command.

Blank lines and leading spaces and tabs are ignored. Lines whose first non-space character is a pound-sign (#) are comments, and are ignored. Note that comments are not allowed on the same line as cron commands, since they will be taken to be part of the command. Similarly, comments are not allowed on the same line as environment variable settings.

An active line in a crontab will be either an environment setting or a cron command. An environment setting is of the form,

name = value

where the spaces around the equal sign (=) are optional, and any subsequent non-leading spaces in value will be part of the value assigned to name. The value string may be placed in quotes (single or double, but matching) to preserve leading or trailing blanks.

Several environment variables are set up automatically by the cron(1) daemon. SHELL is set to /bin/sh, and LOGNAME and HOME are set from the /etc/passwd line of the crontab's owner. HOME and SHELL may be overridden by settings in the crontab; LOGNAME may not.

The LOGNAME variable is sometimes called USER on BSD systems. On these systems, USER will be set also.

In addition to LOGNAME, HOME, and SHELL, cron(1) will look at MAILTO if it has any reason to send mail as a result of running commands in this crontab. If MAILTO is defined (and non-empty), mail is sent to the user so named. If MAILTO is defined but empty (MAILTO=" "), no mail will be sent. Otherwise mail is sent to the owner of the crontab. This option is useful if you decide on /bin/mail instead of /usr/lib/sendmail as your mailer when you install cron — /bin/mail doesn't do aliasing, and UUCP usually doesn't read its mail.

The format of a cron command is very much the V7 standard, with a number of upward-compatible extensions. Each line has five time and date fields, followed by a user name if this is the system crontab file, followed by a command. Commands are executed by cron(1) when the minute, hour, and month of year fields match the current time, and when at least one of the two day fields (day of month, or day of week) match the current time (see Note section). cron(1) examines cron entries once every minute. The time and date fields are:

```
field          allowed values
-----          --------------
minute         0-59
hour           0-23
day of month   0-31
month          0-12 (or names, see below)
day of week    0-7 (0 or 7 is Sun, or use names)
```

A field may be an asterisk (*), which always stands for "first-last". Ranges of numbers are allowed. Ranges are two numbers separated with a hyphen. The specified range is inclusive. For example, 8-11 for an hours entry specifies execution at hours 8, 9, 10 and 11.

Lists are allowed. A list is a set of numbers (or ranges) separated by commas. Examples: `1,2,5,9`, `0-4,8-12`.

Step values can be used in conjunction with ranges. Following a range with `/<number>` specifies skips of the number's value through the range. For example, `0-23/2` can be used in the hours field to specify command execution every other hour (the alternative in the V7 standard is `0,2,4,6,8,10,12,14,16,18,20,22`). Steps are also permitted after an asterisk, so if you want to say "every two hours", just use `*/2`.

Names can also be used for the month `and day of week fields.` Use the first three letters of the particular day or month (case doesn't matter). Ranges or lists of names are not allowed.

The sixth field (the rest of the line) specifies the command to be run. The entire command portion of the line, up to a newline or % character, will be executed by `/bin/sh` or by the shell specified in the `SHELL` variable of the cronfile. Percent-signs (%) in the command, unless escaped with backslash (\), will be changed into newline characters, and all data after the first % will be sent to the command as standard input.

NOTE

The day of a command's execution can be specified by two fields - day of month, and day of week. If both fields are restricted (i.e., aren't *), the command will be run when either field matches the current time. For example, **`30 4 1,15 * 5`** would cause a command to be run at 4:30 am on the 1st and 15th of each month, plus every Friday.

EXAMPLE

```
# use /bin/sh to run commands, no matter what
# /etc/passwd says
SHELL=/bin/sh
# mail any output to `paul', no matter whose crontab this is
MAILTO=paul
#
# run five minutes after midnight, every day
5 0 * * *       $HOME/bin/daily.job >> $HOME/tmp/out 2>&1
# run at 2:15pm on the first of every month
# -- output mailed to paul
15 14 1 * *     $HOME/bin/monthly
# run at 10 pm on weekdays, annoy Joe
0 22 * * 1-5 mail -s "It's 10pm" joe%Joe,%%Where are your kids?%
23 0-23/2 * * * echo "run 23 minutes after midn, 2am, 4am ...,
everyday"
5 4 * * sun echo "run at 5 after 4 every sunday"
```

SEE ALSO

Utility Programs - `cron, crontab`

EXTENSIONS

When specifying day of week, both day 0 and day 7 will be considered Sunday. BSD and ATT seem to disagree about this.

Lists and ranges are allowed to co-exist in the same field. `1-3,7-9` would be rejected by ATT or BSD cron -- they want to see `1-3` or `7,8,9` only.

Ranges can include steps, so `1-9/2` is the same as `1,3,5,7,9`.

Names of months or days of the week can be specified by name.

Environment variables can be set in the `crontab`. In BSD or ATT, the environment handed to child processes is basically the one from `/bin/rc`.

Command output is mailed to the `crontab` owner (BSD can't do this), can be mailed to a person other than the `crontab` owner (SysV can't do this), or the feature can be turned off and no mail will be sent at all (SysV can't do this either).

AUTHOR

Paul Vixie

NAME

`devices` – system device information

SYNOPSIS

`/etc/devices`

DESCRIPTION

The file `/etc/devices` contains lines describing each device normally loaded dynamically into the operating system. The file is read by the utility `devinstall` in response to the **`-a`** option. Each line of `/etc/devices` is of the following form:

info:*driver*:*type*:*extra*

`info` is the pathname of the device information file, `driver` is the pathname of the device driver which controls the device, `type` is either `b` or `c` to indicate whether the device is a block or character device, and `extra` is a decimal value passed to the driver when the device is installed.

NOTE

The devices file does not contain entries for devices configured into the operating system.

SEE ALSO

Utility Programs - `devinstall, devices`

System Calls - `bdv_install(), cdv_install()`

NAME

directory – directory file structure

SYNOPSIS

#include <dir.h>

DESCRIPTION

A directory file is very similar to a regular file, the main difference being that users cannot write into directories. Directories are always an integral number of 512-byte blocks in length. Files are recorded in the directory in `direct` structures, which contain the `inode` number, name length, and name for each file. The name is stored as a null-terminated string, padded to a length divisible by four. The name length value always gives the true string length. Each entry in the directory also includes a record length which is always greater than or equal to the length of the entry. When an entry is removed from a directory as a result of a call to `unlink` or `rename`, the space it occupied is claimed by the entry before it in the file. If an entire block is vacated, it is reclaimed by the operating system.

Directories should always be read via the `directory` routines `opendir`, `readdir`, and friends. These routines return pointers to structures defined in the include file **dir.h**.

```
#define DIRBLKSIZ 512
#define MAXNAMLEN 255
#define MAXPATHLEN 1024
struct direct {
  long d_ino;
  short d_reclen;
  short d_namlen;
  char d_name[MAXNAMLEN+1];
};
typedef struct _dirdesc {
  int d_dfd;
  long d_dloc;
  long d_dsize;
  char d_dbuf[DIRBLKSIZ];
} DIR;
```

Although the nominal size of the `d_name` field of a direct structure is `MAXNAMLEN + 1` bytes, the actual size will be determined by `d_reclen`. The `d_reclen` field gives the actual size of the entire structure, so the actual size of `d_name` is the value of `d_reclen` minus the space consumed by `d_ino`, `d_reclen`, and `d_namlen`.

The details of the `_dirdesc` structure is of importance only to the directory routines themselves.

Every directory contains two special entries, "`.`" and "`..`". The "`.`" entry contains the inode number of the directory itself; it is thus a way of naming the directory when it is the current working directory. The "`..`" entry contains the inode number of the parent of the directory, and thus names the parent when the directory is the working directory. When the directory is the `root` of a mounted file system, however, the "`..`" inode number in the directory will be incorrect, pointing to the same place as "`.`" (that is, it points to the directory itself, since the root of a file system is the top of a tree).

SEE ALSO

Utility Programs - `ls`

Library Functions - `directory()`

NAME

`drivers` – dynamically-loaded device driver table

SYNOPSIS

`/etc/drivers`

DESCRIPTION

The file `/etc/drivers` contains lines describing device drivers normally loaded dynamically into the operating system. The file is read by the utility `drinstall` in response to the **`-a`** option. Each line of the file is of the following form:

driver:*type*

`driver` is the file name of the device driver software, and `type` is either `b` or `c` to specify whether the driver is block or character.

The `type` field is `c` for character device drivers, `b` for block device drivers, and `m` for STREAMS modules.

NOTE

The `drivers` file does not contain information about drivers configured into the operating system.

SEE ALSO

Utility Programs - `drinstall, drivers`

System Calls - `dr_install()`

NAME

`ethers` – Ethernet address to hostname database

DESCRIPTION

The `ethers` file contains information regarding the known (48 bit) Ethernet addresses of hosts on the Internet. For each `host` on an Ethernet, a single line should be present with the following information:

ethernet_address official_hostname

Separate items by any number of blanks and/or TAB characters. A `#` indicates the beginning of a comment extending to the end of line.

The standard form for Ethernet addresses is `x:x:x:x:x:x` where `x` is a hexadecimal number between `0` and `ff`, representing one byte. The address bytes are always in network order. Host names may contain any printable character other than a space, tab, newline, or comment character. It is intended that hostnames in the `ethers` file correspond to the hostnames in the `hosts` file.

EXAMPLE

The following is a sample `/etc/ethers` file:

```
8:0:1b:0:a0:17 dg1
0:0:77:1a:0:6a sales
8:0:20:0:a7:5d sun1
```

FILES

`/etc/ethers`

SEE ALSO

File Formats - `hosts`

NAME

`exports, xtab` – directories to export to NFS clients

SYNOPSIS

/etc/exports
/etc/xtab

DESCRIPTION

The `/etc/exports` file contains entries for directories that can be exported to NFS clients. This file is read automatically by the `exportfs` command. If you change this file, you must run `exportfs` for the changes to affect the daemon's operation.

The `/etc/xtab` file contains entries for directories that are currently exported. Use the `-u` option of `exportfs` to remove entries from this file.

An entry for a directory consists of a line of the following form:

directory option[, *option*]...

directory is the pathname of a directory. *option* is one of:

`ro` — Export the directory read-only. If not specified, the directory is exported read-write.

`rw=`*hostname*[:*hostname*]..
Export the directory read-mostly. Read-mostly means read-only to most machines, but read-write to those specified. If not specified, the directory is exported read-write to all.

`anon=`*uid* — If a request comes from an unknown user, use *uid* as the effective user ID. Note: root users (`uid 0`) are always considered “unknown” by the NFS server, unless they are included in the `root` options below. The default value for this option is `-2`. Setting the value of `anon` to `-1` disables anonymous access.

`root=`*hostname*`[:`*hostname*`]..`
: Give `root` access only to the root users from a specified *hostname*. The default is for no hosts to be granted `root` access.

`access=`*hostname*`[:`*hostname*`]...`
: Give `mount` access to each *hostname* listed. Each *hostname* in the list is checked in the `/etc/hosts` database. The default value allows any machine to `mount` the given directory.

A "`#`" (pound-sign) anywhere in the file indicates a comment that extends to the end of the line.

EXAMPLE

```
/usr  access=clients  #export to my clients
/usr/local   #export to the world
/usr2  access=lynx:lynxbsd:topcat #export only to these machines
/usr/sun root=lynx:lynxbsd #give root access only to these
/usr/new anon=0 #give all machines root access
/usr/bin ro  #export read-only to everyone
/usr/stuff access=lynx,anon=-3,ro #several options on one line
```

FILES

`/etc/exports`

`/etc/xtab`

`/etc/hosts`

`/net/rc.network`

SEE ALSO

Utility Programs - `exportfs, nfsd`

File Formats - `hosts`

NAME

fstab, mtab – table of file systems

SYNOPSIS

/etc/fstab

/etc/mtab

```
#include <fstab.h>

#include <mtab.h>
```

DESCRIPTION

The file /etc/fstab contains lines describing each file system normally mounted on the directory structure. The file is read by the mount and umount utilities in response to the **–a** option. The file /etc/mtab is created and maintained by mount and umount to reflect the file systems actually mounted at any one time. The /etc/fstab may contain entries for mounting block and NFS type file systems.

Each line of /etc/fstab for block device mounting is of the following form:

device:*dir*:*type*:*freq*:*passno*

where *device* is the block device file name that contains the file system, *dir* is the name of the directory where the file system is mounted, and *type* is the file system type (see below). The fields *freq* and *passno* are included for compatibility reasons but are ignored by LynxOS utilities.

Each line of /etc/fstab for NFS mounting is of the following form:

hostname:*exported-directory*:*local-directory*:*options* where *hostname* is the NFS server's host name, *exported-directory* is the directory name of the exported directory in the NFS server, and *local-directory* is the directory name on which the NFS mounting happens. Supported options for NFS mounting are described in the “Options” section below.

The `getfsent` routines return pointers to `fstab` structures, defined as follows in the `fstab.h` file:

```
struct fstab {
  char *fs_spec;  /* Block device name */
  char *fs_file;  /* Directory name (mount point) */
  char *fs_type;  /* One of the FSTAB_ constants */
  int fs_freq;    /* Not used by LynxOS */
  int fs_passno;  /* Not used by LynxOS */
};

#define FSTABRW    "rw"  /* Read and write */
#define FSTABRO    "ro"  /* Read only  */
#define FSTABXX    "xx"  /* Ignore entry */
```

The `FSTAB_ constants` indicate whether the file system is to be mounted read-only or read and write. The `XX` value causes an entry in `/etc/fstab` to be ignored by the utility programs.

The `/etc/mtab` file is not human-readable. It is a collection of `mtab` structures, as defined in `mtab.h`:

```
struct mtab {
  char m_path[1025];   /* Where mounted */
  char m_dname[1025];  /* Device name */
  char m_type[4];      /* Access mode */
  char m_opts[1024];   /* options for NFS mounting */
};
```

The `mtab` file is completely maintained by `mount` and `umount` to reflect the currently mounted file systems. The file is deleted upon system restart to avoid garbage or duplicate entries.

OPTIONS

The following options are supported for NFS mounting entries in `/etc/fstab`. Each option is to be separated by comma.

`ro/rw`	Mount the filesystem with read-only or read-write access.
`retry=`*n*	The number of times to retry the `mount` operations, in case of soft mounting.
`rsize=`*n*	The read buffer size is set to *n*.
`wsize=`*n*	The write buffer size is set to *n*.
`timeout=`*n*	Set the NFS timeout to *n* tenths of a second (*n**100 msec).
`retrans=`*n*	The number of NFS re-transmissions.
`port=`*n*	The server IP port number.
`soft/hard`	Return an error if the server fails to respond or continue the retry request until the server responds.
`intr`	Allow keyboard interrupts on hard mounts.
`acregmin=`*n*	Hold the cached attributes for at least *n* seconds after modifications.
`acregmax=`*n*	Hold cached attributes for no more than *n* seconds after file modifications.
`acdirmin=`*n*	Hold cached attributes for at least *n* seconds after directory update.
`acdirmax=`*n*	Hold cached attributes for at most *n* seconds after directory update.
`actimeo=`*n*	Set `min` and `max` times for regular files and directories to *n* seconds.
`nodc`	Suppress data caching.

EXAMPLES

Block type device entries.

```
/dev/sd0a:/:rw:0:0
/dev/sd0b:/mnt-sd0b:ro:0:0
/dev/sd0c:/mnt:rw:0:0
```

NFS type mount entries.

```
topcat:/usr/home:/myhome:ro,rsize=0x2000,wsize=0x2000
lynx:/usr/tmp:/mytmp:rw,nodcache
```

SEE ALSO

Utility Programs - `mount`, `umount`, `df`

System Calls - `mount()`, `umount()`

Library Functions - `getfsent()`

NAME

`gateways` – gateway information for `routed`

SYNOPSIS

`/etc/gateways`

DESCRIPTION

The file `/etc/gateways` describes gateways for `routed`. When `routed` is started up, it reads the `/etc/gateways` to find gateways which may not be located using only information from the `SIOGIFCONF ioctl`. Refer to the `routed` manual page for more details.

`/etc/gateways` is comprised of a series of lines, each in the following format:

`<net|host>` *name1* `gateway` *name2* `metric` *value* `\`
`<passive|active|external>`

The net or host keyword indicates if the route is to a network or specific host.

name1 is the name of the destination network or host. This may be a symbolic name located in `/etc/networks` or `/etc/hosts` or, if `routed` is started after `named`, a symbolic name known to the name server or an Internet address specified in Internet dot notation.

name2 is the name or address of the gateway to which messages should be forwarded.

value is a metric indicating the hop count to the destination network or host.

One of the keywords `passive`, `active`, or `external` indicates if the gateway should be treated as passive or active (see `routed`), or whether the gateway is external to the scope of the `routed` protocol.

SEE ALSO

Utility Programs - `named, routed`

File Formats - `hosts, networks`

NAME

`group` – group information file

SYNOPSIS

/etc/group

#include <grp.h>

DESCRIPTION

The file `/etc/group` contains a list of group names, numbers, and members. It is read by the `getfsent` routines.

Each line of `/etc/group` is of the form:

name:*passwd*:*gid*:*user-list*

name is a name for this `group`. It is used by programs to translate a group ID number into a recognizable name. *passwd* is unused and exists for forgotten historical reasons. The group ID number is supplied at *gid*, and the list of user names belonging to the `group` *name*, follows at *user- list*. Commas should be inserted to separate names of users.

When the operating system determines accessibility to a file, it checks the effective user ID against the file owner first. If the user ID mismatches, then the effective group ID and all groups to which the current effective user ID belongs are checked against the group owner of the file. The list of groups is usually computed and established by `login`.

The `getgrent` routines retrieve information from `/etc/group` and return it in the form of `group` structures, defined in `grp.h`.

```
struct group {
  char *gr_name;
  char *gr_passwd;
  int gr_gid;
  char **gr_mem;
};
```

SEE ALSO

System Calls - `setgroups(), getgroups(), getgid()`

Library Calls - `buildgroups()`

NAME

`hosts` – host name data base

DESCRIPTION

The `hosts` file contains information regarding the known hosts on the DARPA Internet. For each host a single line should be present with the following information:

Internet-address official-host-name aliases

Items are separated by any number of blanks and/or tab characters. A “`#`” indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file. This file is normally created from the official host data base maintained at the Network Information Control Center (NIC), though local changes may be required to bring it up to date regarding unofficial aliases and/or unknown hosts.

Network addresses are specified in the conventional “`.`” notation using the `inet_addr` routine from the Internet address manipulation library. Host names may contain any printable character other than a field delimiter, newline, or comment character.

NOTE

A name server should be used instead of a static file. A binary indexed file format should be available for fast access.

FILES

/etc/hosts

SEE ALSO

Library Functions - `gethostent()`

NAME

`hosts.equiv` – hostnames with "equivalent" user IDs

DESCRIPTION

This file is used by `rcp`, `rsh`, and `rlogin` to determine a set of hosts whose users are considered to be "equivalent" to the users of the same name on this system. In an environment where a single organization might have many systems used by a common set of users, it is often the case that a single user will have the same ID on many different systems. The use of `/etc/hosts.equiv` to specify the list of systems with common users, and common user IDs, allows users access to the local system from each of these remote systems without requiring passwords. Further, `rsh` and `rcp` require that the user be considered equivalent on the local and remote systems, or the operation is rejected.

`rshd`, used to support `rsh` and `rcp` requests, uses `/etc/hosts.equiv` in the following way. When the connection is made, `rshd` gets the ID of the user on the remote ("calling") system. It then looks up the remote user ID in the local `/etc/passwd` file. If the remote user is not the super-user, then `/etc/hosts.equiv` is checked for the name of the remote host. If it is found, the user is considered to be equivalent to the user of the same local name, and the command proceeds. If the host name is not found, or if the remote user is the super-user, then `rshd` checks the file `.rhosts` in the login directory found in `/etc/passwd`. If an entry is found for the remote host, or for this local user name and remote host combination, then the user is considered equivalent and the command proceeds. If this test fails, the command is terminated. `rlogin` uses these files in an analogous fashion.

The format of `/etc/hosts.equiv` is a list of host names one per line. The "primary" host name (i.e. the first one in `/etc/hosts`) must be used.

FILES

`/etc/hosts.equiv`

SEE ALSO

Utility Programs - `rcp, rlogin, rsh, rlogind, rshd`

File Formats - `rhosts`

NAME

`inetd.conf` – Internet servers database networking

DESCRIPTION

The `inetd.conf` file contains the list of servers that inetd invokes when it receives an Internet request over a socket. Each server entry is composed of a single line of the form:

service-name socket-type protocol wait-status uid
server-program server-arguments

Fields can be separated by either spaces or TAB characters. A "#" (pound-sign) indicates the beginning of a comment; characters to the end of the line are not interpreted by routines that search this file.

service-name Is the name of a valid service listed in the file `/etc/services`. For RPC services, the value of the service-name field consists of the RPC service name, followed by a slash and either a version number or a range of version numbers (for example, mountd).

socket-type Can be one of:

`stream`	for a stream socket.
`dgram`	for a datagram socket.
`raw`	for a raw socket.
`rdm`	for a "reliably-delivered message" socket.
`seqpacket`	for a sequenced packet socket.

protocol Must be a recognized protocol listed in the file `/etc/protocols`. For RPC services, the field consists of the string `rpc` followed by a slash and the name of the protocol (for example, `rpc/udp` for an RPC service using the UDP protocol as a transport mechanism).

wait-status Is `nowait` for all but "single-threaded" datagram servers - servers which do not release the socket until a time-out occurs (such as `talkd`). These must have the status

	`wait`. Although the `tftpd` establishes separate "pseudo-connections", its forking behavior can lead to a race condition unless it is also given the status `wait`.
uid	Is the user ID under which the server should run. This allows servers to run with access privileges other than those for `root`.
server-program	Is either the pathname of a server program to be invoked by `inetd` to perform the requested service, or the value `internal` if `inetd` itself provides the service.
server-arguments	If a server must be invoked with command-line arguments, the entire command line (including argument 0) must appear in this field (which consists of all remaining words in the entry). If the server expects `inetd` to pass it the address of it's peer (for compatibility with 4.2BSD executable daemons), then the first argument to the command should be specified as "`%A`".

FILES

`/etc/inetd.conf`

`/etc/services`

`/etc/protocols`

SEE ALSO

Utility Programs - `services, comsat, inetd, talkd, tftpd`

NAME

inode – on-disk inode format file

SYNOPSIS

#include <disk.h>

DESCRIPTION

An inode ("eye-node") is a structure which contains attributes and disk indices to uniquely identify a file. Inodes are referenced by number in directory files. They are packed four per block in a reserved contiguous space in the file system, starting at block number 2. The format of an inode is described by the d_inode structure defined in **disk.h**:

```
struct d_inode {
  short i_mode;    /* protection */
  short i_nlink;  /* number of hard links to this file */
  long i_genid;   /* NFS generation count */
  unsigned short i_uid;        /* user id */
  unsigned short i_guid;      /* group id */
  long i_size;    /* file size (in bytes) */
  long i_atime;   /* time file data was last accessed */
  long i_mtime;   /* time when file data was last modified */
  long i_ctime;   /* time when file status was last changed */
  long i_blksize; /* actual number of blocks used in file */
  char i_blocks[3*13]; /* direct and indirect blocks*/
};
```

The i_blocks array contains 13 24-bit block addresses. The first 10 addresses point to the first 10 data blocks of the file. The 11th address points to an indirect block, which is a disk block containing 128 32-bit disk addresses. The addresses in the first indirect block point to the 11th through 138th data blocks of the file. The 12th address in the inode points to a double-indirect block, whose addresses point to 128 indirect blocks, whose addresses in turn point to the 139th through 16,522nd data blocks. Finally, the 13th and last address in the inode points to a triple-indirect block, which points to 128 double-indirect blocks, which point to 16,384 indirect blocks, which point to the 16,523rd through 2,113,674th data blocks.

Information stored in an inode is usually accessed through the `stat` system call. Only very special utilities, such as `fsck`, ever directly manipulate actual inodes.

SEE ALSO

Utility Programs -	`df, fsck, mkfs`
System Calls -	`stat()`
File Formats -	`superblock`

NAME

`insttab` – table of dynamically-installed drivers and devices

SYNOPSIS

`/etc/insttab`

DESCRIPTION

The file `/etc/insttab` contains lines describing drivers to be dynamically installed into the operating system, devices to be installed for each driver, and special files (i.e. nodes) to be created for each device. The `dynaminst` utility reads this file by default if not explicitly given another file. This file is known as the installation file.

Each entry of the installation file is a line of the form:

type:*field-list*

Where *type* is a single character indicating the entry type, and *field-list* is a colon-separated list of values appropriate to the particular entry type. Entries may not be split across multiple lines, but they may contain spaces or tabs.

The entry types of the installation file are as follows:

`C` Character Device Driver Installation:

`C`:*path* — *path* specifies the pathname of the driver object file that is to be dynamically installed into the operating system.

`B` Block Device Driver Installation:

`B`:*path* — *path* specifies the pathname of the driver object file that is to be dynamically installed into the operating system.

`D` Major Device Installation:

`D`:*path*:*driver-id*:*raw-partner*

path specifies the pathname of the device info object file that is to be dynamically installed into the operating system. *driver-id* is the ID of the driver under which this device is being installed. In most cases, this field will be empty, meaning the device is to be installed under the most recently-listed device driver. If *driver-id* is not empty, then the referenced driver must have been configured into the operating system before the `dynaminst` utility was invoked. *raw-partner* is meaningful only for block devices. It should be set to the pathname of the character major device which provides a "`raw`" interface to the same physical device as the block entry.

N Node Creation:

`N:`*path*`:`*minor-number*

path specifies the pathname of the node that is to be created. *minor-number* is combined with the major number of the most recently listed device to form the node device number. Both fields must be present.

The order of the entries in the file dictates which devices correspond to which drivers, and which nodes correspond to which devices. The first entry in the installation file should be a driver entry. After the driver should come the first major device, then each of the minor devices for that major device. More major device and node lists may follow. The sequence then repeats for the next driver. Thus the configuration file takes the form:

```
device driver entry
device entry
node entry
node entry
...
device entry
node entry
node entry
...
device driver entry
device entry
node entry
node entry
...
```

and so on. Blank lines and lines beginning with “#” are ignored.

The `dynaminst` utility takes action on each line of the installation file as it is read. Thus, a block major device with a raw partner must be listed <u>after</u> the referenced character major device. Driver IDs are assigned positionally; at the time of installation, the lowest available unused ID is chosen. The same is true for major device numbers. Device types (character or block) for nodes are implied by driver association.

EXAMPLE

```
# Example of an installation file containing 3 drivers
 C:/usr/test/mydrvr.o
  D:/usr/test/myinfoA.o::
    N:/usr/test/mydevA0:0
    N:/usr/test/mydevA1:1
    N:/usr/test/mydevA2:2
    N:/usr/test/mydevA3:3
  D:/usr/test/myinfoB.o::
    N:/usr/test/mydevB0:0
```

```
    N:/usr/test/mydevB1:1
    N:/usr/test/mydevB2:2
    N:/usr/test/mydevB3:3

C:/sys/drivers/new/newcdrvr.o
  D:/sys/devices/newcinfo.o::
    N:/dev/rnewdev:0

B:/sys/drivers/new/newbdrvr.o
  D:/sys/devices/newbinfo.o::/sys/devices/newcinfo.o
    N:/dev/newdev:0
```

NOTE

The installation file is similar to the configuration table used by the `config` utility. However, they differ in syntax and usage and should not be confused. The installation file does not contain information about drivers or devices configured into the operating system.

SEE ALSO

Utility Programs -	`config, devinstall, devices, drinstall, drivers, dynaminst, mknod`
System Calls -	`bdv_install(), bdv_uninstall(), cdv_install(), cdv_uninstall(), dr_install(), mknod()`
File Formats -	`config`

NAME

`kconfig` – kernel configuration file format

DESCRIPTION

The LynxOS operating system can be configured to include all the features available in LynxOS, or just those that are needed. `kconfig` and the `KERNEL.CFG` file make it possible to prevent certain features from being included in the LynxOS kernel.

In order to remove a feature its object file must be replaced with a `stub` file. The `KERNEL.CFG` file contains a list of object files that are to be replaced with `stub` files. It may also contain a list of object files that are not to be replaced. If an object file is to be replaced with a `stub` file, `KERNEL.CFG` must contain a line like this:

–*module*

The minus sign in the last example should be replaced by a plus sign if an object file should not be replaced by a `stub`. Any object module not listed in the `KERNEL.CFG` file will not be replaced.

FILES

`KERNEL.CFG`

SEE ALSO

Utility Programs - `kconfig`

NAME

`keyb` – keyboard layout file for use with `keyb` utility

DESCRIPTION

The `keyb` keyboard layout file is an ASCII file which tells the codes the keyboard driver shall generate when a key is hit. The name of a `keyb` file usually indicates the country which the keyboard refers to; e.g., `fr`, `gr`, `uk`, `us` etc. for France, Germany, United Kingdom, USA, respectively. The `keyb(1)` utility looks for this file in the default directory `/usr/local/lib/keyb/`.

The `keyb` keyboard layout file is read by the `keyb` utility (see `keyb(1)`), which translates the file to a binary table and passes the result to the `atc` driver (see `atc(4)`).

The following sections discuss the syntax and semantics of the `keyb` keyboard layout file in detail.

The Sections of the Keyboard Layout File

A `keyb` keyboard layout file consists of up to 6 sections, each section must be introduced by a special line which consists of the keyword `$LABEL` followed by at least 1 space and a unique label term:

```
$LABEL normal
$LABEL shift
$LABEL caps
$LABEL control
$LABEL alt-right
$LABEL compose
```

The sequence of the sections is arbitrary. The sections labeled `normal`, `shift`, `caps`, `control`, and `alt-right` are called Mapping Sections. They refer to the various keyboard modes which are controlled by the meta-keys (**Shift**, **Caps**, **Ctrl**, **Alt**, where **Alt**-**Right** refers to the right-hand **Alt** key). A simple geographic mapping is used for assigning individual character codes to the keys. See “The Mapping Sections” for more information.

The `compose` section is different. It tells the character codes which are generated (composed) by striking two keys; e.g., an **E** with an accent could be generated by striking the single quote followed by the letter **E**.

The **Alt**-**Right** and `compose` sections are optional, the other sections are mandatory. If the **Alt**-**Right** section is missing then the right-hand **Alt** key will be automatically the same as the left-hand **Alt** key. The translation table for the left-hand **Alt** key is defined internally in the ATC console driver and can't be affected from outside.

The Mapping Sections

The sections labeled `normal`, `shift`, `caps`, `control`, and `alt-right` are quite similar to each other: the "body" of each section consists of 4 lines following the `$LABEL` line. The 4 lines refer to the 4 rows of the keyboard's main field: i.e. the white keys which designate letters, digits and separators. The **Space Bar**, **Back Space**, **Enter**, **Tab**, **Esc**, Function Keys, and keypad are not involved.

The Rows of a Mapping Section

Within the "body" of a mapping section, each line starts with a predefined tag. The tag words refer to the rows of the keyboard's main field:

```
row1 (top row, normally digits etc.)
row2
row3
row4 (bottom row, just above the space bar)
```

Within a section, the sequence of the lines is arbitrary since the attachment of each line is defined by its tag rather than by its position. Each tag is followed by at least 1 space and a so-called description of the respective row of the keyboard. Essentially, the description is a simple string defining the code values for the keys in that row. The code values may be defined directly using printable characters or by means of special coding sequences. Sequences are most often used for expressing those character codes which can't be represented as printable characters. The syntax of the sequences is explained in "Special Codes and Sequences" below.

The characters and sequences of a description string are mapped to the keys by their natural order: the first code is attached to the left-most key in the respective row etc. This way the description strings define specific code values for each key of the keyboard's main field. The `atc` driver will deliver this code when the respective key is hit.

Comments may be applied wherever they can't be confused with the essential information: a comment may be appended to any complete line by using an intermediate space character for separation. Furthermore, empty lines and lines starting with an asterisk (`*`) in column 1 will be completely ignored.

The Keyboard Type

The `keyb` utility supports two basic types of PC/AT keyboards which differ with respect to the number of keys in the main field:

Row Number	US-Type Layout with 101 Keys	Int'l Layout with 102 Keys
#1	13 keys	13 keys
#2	13 keys	12 keys
#3	11 keys	12 keys
#4	10 keys	11 keys

The `keyb` utility recognizes the keyboard type simply by the size of its rows; i.e., by counting the keys defined in the various description lines. If a section does not fit to one of the above keyboard types or if the various sections do not refer all to the same keyboard type then `keyb` will abort with an error message.

Special Properties of the Individual Mapping Sections

Please note at this point, that `+`, `-`, and `\` have special meanings within the description strings. They are discussed in paragraph 4.

Within the `normal` and `shift` sections, the characters and sequences of the description strings define directly the code value which will be passed to the system when the related key is hit.

The `alt-right` section is evaluated in a somewhat different way: first, the built-in ALT table (which belongs to the left-hand ALT key and is not accessible to the user) is copied to the internal `alt-right` table. Then, the `alt-right` section data are overlaid. Thus, if the `alt-right` section is omitted, the right and left ALT keys will become effectively the same.

In the `control` section, the description strings will be interpreted in a different way: all characters in the range from `@` to _ (i.e., `0x40..0x5f`) will automatically be converted to the respective control codes (`0x00..0x1f`). This allows a convenient notation of the typical control codes without using sequences. On the other hand, sequences (or any character preceded by the introducer `\`) will *not* undergo the automatic conversion. This means that **Ctrl-** cannot be encoded by `\`. Instead, a sequence must be used ("`\x1c`" or "`\d028`").

The `caps` section determines which keys shall be affected from the keyboard's **Caps Lock** state. Essentially, if the keyboard is run with **Caps Lock On** the `normal` and the shifted codes will be interchanged for those keys mapped as '1' in this section. The keys mapped as 0 remain unaffected from the **Caps Lock** state.

The Compose Section

What Is Autocomposing?

A convenient means for generating special characters with accents, etc. are the so-called autocomposing keys. This means that there are some special keys which don't give an immediate response after being hit. Instead, the device driver software waits for the next keystroke and then—if the pair of keystrokes matches some internal table—a special code (the "composed" character) is delivered. If no match was found then two codes are delivered which correspond to the original codes of the two keys. As an example, consider the french "circumflex":

First Key	Second Key	Effect Char(s)	Notes
^	a	<a with ^>	typical composing
^	^	<not yet>	double autocompose keystroke is effectively the same as a single stroke

First Key	Second Key	Effect Char(s)	Notes
^	x	^x	no match found - both codes are delivered unchanged
^	<space>	^	<space> forces the original code of the first key to be delivered

Declaring the Autocompose Capability of a Key

The autocompose capability of a key must be declared in the respective Mapped Section of the Keyboard Layout File by inserting `\C` just before the definition of that key. Please note that `\C` does not define a new key! Instead, it indicates that the next key within this row will have the autocompose capability. There are some facts to be known:

- within the `caps` section, `\C` is meaningless
- within any Mapped Section, `\C` is meaningless before `+`, `-`, and `\C`
- otherwise, `\C` means literally `C`

Defining the Compose Matching List

For each key which was declared having the autocompose capability the `compose` section must provide a matching list. The matching list is a simple string starting with the autocompose key itself followed by pairs of characters, each pair consisting of the second key and the corresponding "composed" output character.

Each matching list must be introduced by a tag name. The tag names may be chosen arbitrarily except that a tag name must be unique and it must not be empty and must not contain any spaces or tabs or other non-printing characters.

Example: Autocompose for US-type Keyboard

```
$LABEL normal
row1 \C`1234567890-=   "`" is autocompose key
row2 qwertyuiop[]\\
row3 asdfghjkl;\C'     "'" is autocompose key
row4 zxcvbnm,./
$LABEL shift
row1 \C~!@#$%\C^&*()_+ "~" and "^" are autocompose keys
.
.
.
$LABEL compose !!! ISO Latin-1 codes for special chars
* composing "~" with "n" and "N" tilde "~n\xf1N\xd1"
* composing "`" with "a", "e", "i", "o", and "u" accent-grave
"`a\xe0e\xe8i\xeco\xf3u\xf9"
* composing "'" with "a", "e", "i", "o", "u", and "E" accent-
aigue "'a\xe1e\xe9i\xedo\xf4u\xfaE\xc9"
* composing "^" with "a", "e", "i", "o", and "u" circumflex
"^a\xe2e\xeai\xeeo\xf5u\xfb"
```

Special Codes and Sequences

Plus Sign - Skip This Key

A plus character indicates that the output of the respective key shall be left unchanged in the current section. This code is very useful for the **Alt-Right** section where only a few keys are to be changed. If the ASCII code for plus (`0x2b`) shall be output then a sequence introducer (\) must precede the plus sign.

This code does not have any special meaning in the `compose` and `caps` sections.

Minus Sign - Disable This Key

A minus character indicates that the respective key shall be disabled in the current section. If the ASCII code for minus (`0x2d`) shall be output then a sequence introducer (\) must precede the minus sign. This code does not have any special meaning in the `compose` and `0` sections.

The following table describes the backslash—the sequence introducer.

Sequence Introducer	Description	ASCII
“\x<h>”	Arbitrary 2-digit hex number <h> Example: \x13 = ASCII DC3 (**Ctrl-S**) If “\x” is *not* followed by 2 hex digits, “\x” translates to x	n/a
\d<d>	Arbitrary 3-digit number <d>, will be masked to 8 bits Example: \d025 = ASCII EM (**Ctrl-Y**) If \d is *not* followed by 3 hex digits, \d translates to d	n/a
\C	Enable the Autocompose Capability for the next key (see “The Compose Section”; meaningless in the compose and caps sections)	n/a
\+	Plus	0x2b
\-	Minus	0x2d
\0	ASCII NULL	0x00
\n	Newline code	0x0a (usually)
\r	Carriage Return code	0x0d
\t	Tab code	0x09
\b	Backspace code	0x08
\f	Formfeed code	0x0c
\\	Backslash	0x5c
\<cr>	At end-of-line, a single \ suffices for encoding	0x5c
\<c>	Any other character following \ will be taken literally	n/a

FILES

/usr/local/doc/atc-4X-chartab

DIAGNOSTICS

The error messages returned from `keyb` are pretty verbose and self-explanatory.

SEE ALSO

Utilities - `keyb`

Device Facilities - `atc`

NAME

`libr` – library file format

SYNOPSIS

```
#include <ar.h>
#include <ranlib.h>
```

DESCRIPTION

The include file **ar.h** (not `libr.h`, unfortunately, for compatibility reasons) contains a description of the header block prepended to the contents of each component file in a library. A library is simply a collection of files, usually object files, with the above mentioned header blocks and an identifying "magic string" at the beginning. Libraries are created and maintained by the `libr` utility.

```
#define ARMAG "!<arch>\n " /* At start of lib */
#define SARMAG 8             /* Length of magic thing */
#define ARFMAG "`\n "        /* To verify that header is nice */
struct ar_hdr {
  char ar_name[16];    /* File name */
  char ar_date[12];    /* Last modification date */
  char ar_uid[6];      /* User ID */
  char ar_gid[6];      /* Group ID */
  char ar_mode[8];     /* File mode (in octal) */
  char ar_size[10];    /* File size */
  char ar_fmag[2];     /* Magic string (ARFMAG) */
};
```

When `libr` creates or modifies a library, it creates a symbol table for all of the object files within it. The symbol table is a normal library entry which is inserted at the front of the library. The format of the symbol table, called `__SYMDEF`, is described in the include file **ranlib.h**.

```
struct ranlib {
  union {
    off_t ran_strx;  /*disk version-index into strings*/
    char *ran_name;  /*in-core version-point to string*/
  } ran_un;
  off_t ran_off;    /*position in lib file of object file*/
};
```

The symbol table consists of a `long` value giving the number of bytes of `ranlib` structures which follow, the `ranlib` structures themselves, a long giving the number of bytes of strings, and finally the null-terminated strings themselves. The `ran_strx` field in each `ranlib` structure gives an index into the string table, with position 0 at the first string (not the string table size longword). The `ran_off` field gives the file position in the library of the beginning of the header (library header, not object header) for the object file containing the symbol.

Luckily, the linker and librarian are the only entities which really need concern themselves with the above details.

SEE ALSO

Utility Programs - `cc, libr`

NAME

log – log data file format

SYNOPSIS

/usr/spool/ssp/log
#include <log.h>

DESCRIPTION

The log data file is the output of the read_log utility. It contains a sequence of records, terminated by a status record. The record type depends on the log it was read from, which should be clear from the name of the data file. The record declarations are in **log.h**. read_log appends data to the end of the file, so there may be multiple status records. The size of the status record will be the same as the other records in the file. The format of the status record is as follows:

```
struct stat_entry {
   int sec;         /* second when entry made */
   int usec;        /* micro-second when entry made */
   int log_id;      /* dummy */
   int datalost;    /* log status flags */
 };
```

The log_id field can be ignored; the datalost field is set by one or more of the following flags:

LOG_DISABLED	The log is disabled.
LOG_ENABLED	The log is enabled.
HAVE_MORE	The log was not emptied when it was read.
DATA_LOST	Data from the log was lost since the previous read.

Data may be lost from the log when it is not read frequently enough. set_read_limit may be used to decrease the possibility of data loss.

FILES

/etc/faultlog	System fault log.

/etc/fslog	Illegal file access log.
/etc/ptimelog	Process time log.
/etc/syscalllog	System call log.
/usr/spool/ssp/log/[logname]	Default log output file.

SEE ALSO

Utility Programs - `disable_log`, `enable_log`, `get_log_size`, `set_log_size`, `display_log`, `read_log`, `set_read_limit`

NAME

`mkimage.spec` – specification file for the `mkimage` utility

DESCRIPTION

The specification file for the `mkimage` utility can have any valid file name and may contain lines with the following keywords. Default values are shown in parentheses ().

Designates a comment from the # to the end of the current line.

target=[x86|ppc|68k|sparc]
Designates the targeted system. May be one of: *x86*, *ppc*, *68k*, or *sparc*.

osstrip=[local|all|none]
Causes local symbol definitions to be stripped from the kernel text file. May be omitted (**none**) or:

none	no symbols are stripped
local	strips local symbols only
all	strips all the symbols

ostext=[ram|rom]
Designates where the kernel will reside in the running system. May be omitted (**ram**), or have a value of either *ram* or *rom*.

elf=[true|false]
Generates ELF statements for netbooting by the PowerPC OpenBoot Firmware. May be omitted (**false**) or have a value of either *true* or *false*.

rom= Designates the starting address of the ROM.

flags= Designates the boot flags to be passed to the kernel.

kernel=path Designates the path and name of the kernel text file.

nodetab=path Designates the path and name of the `nodetab` used to generate the `/dev` entries for the kernel. If omitted, the path for the kernel is used.

resident=[true|false]
Designates whether or not the application programs will be resident in memory. May be omitted (**false**) or have the value of either *true* or *false*.

text=[ram|rom] Designates the resident application programs will be in ROM or moved to RAM. May be omitted (**ram**), or have a value of either *ram* or *rom*.

strip=[true|false]
Designates whether or not the application files will have symbols stripped before being placed into the ROM kit file. May be omitted (**false**) or have the value of either either *true* or *false*.

root=[ram|rom] Designates the root file system will be resident in either read only or random access memory. May be omitted (**ram**), or have a value of either *ram* or *rom*.

free= Designates the number of free blocks to be generated for the file system.

inodes= Designates the number of free inodes to be generated for the file system.

directory=path Designates a target file system directory. It must be a fully qualified path name.

recurse=path Designates a fully qualified path name to a source file system directory. All the files contained in that directory as well as the subdirectories and files contained in the subdirectories are placed in the target file system.

use=path Designates a fully qualified path name to a source file system directory. All the files contained in that directory are placed in the target file system.

file= Designates a target file name. The file is created in directory specified in the immediately preceding **directory=** specification line.

source= Designates a fully qualified path name to a source file to be copied into the target file system as the file specified in the **file=** specification line.

mode= Designates the file mode for the target file.

owner= Designates the numeric owner ID of the target file. No check is made to assure the ID is valid.

group= Designates the numeric group ID of the target file. No check is made to assure the ID is valid.

link name1 name2
Designates a hard link of **name1** to **name2**.

symlink name1 name2
Designates a symbolic link of **name1** to **name2**.

Lines containing the **directory** keyword may have the following format and must be in a contiguous line within the specification file:

```
directory=path1 use=path2 [mode=filemode] [owner=owner] [group=group]

directory=path1 recurse=path2 [mode=filemode] [owner=owner] [group=group]

directory=path1 [mode=filemode] [owner=owner] [group=group]
```

Lines containing the **file** keyword have the following format and must be in a contiguous line within the specification file:

```
file=name1 source=name2 [mode=filemode] [owner=owner] [group=group]
```

The **mode=**, **owner=** and **group=** are all optional and may be omitted for both the file and directory specification lines. When they are omitted, they default to the source file or path attributes.

EXAMPLE

This example demonstrates a sample `mkimage.spec` file for a network bootable (or bootable) LynxOS ROM kernel image for SPARC.

```
# "Normal rom kitted kernel"
# Kernel
    target=sparc osstrip=local ostext=ram
    kernel=/sys/lynx.os/a.out
    nodetab=/sys/lynx.os/nodetab

# File system
    free=500 inodes=10 root=ram

# Text files:
    strip=true #text=ram resident=true

    directory=/
        file=init               source=/init

    directory=/bin
        file=cat                source=/bin/cat
        file=cp                 source=/bin/cp
        file=date               source=/bin/date
        file=devices            source=/bin/devices
        file=df                 source=/bin/df
        file=dirname            source=/bin/dirname
        file=drivers            source=/bin/drivers
        file=du                 source=/bin/du
        file=echo               source=/bin/echo
        file=false              source=/bin/false
        file=fsck               source=/bin/fsck
        file=kill               source=/bin/kill
        file=less               source=/bin/less
        file=lesskey            source=/bin/lesskey
        file=ln                 source=/bin/ln
        file=ls                 source=/bin/ls
        file=mkdir              source=/bin/mkdir
        file=more               source=/bin/more
```

```
        file=mount          source=/bin/mount
        file=ps             source=/bin/ps
        file=pwd            source=/bin/pwd
        file=rc             source=/bin/rc
        file=reboot         source=/bin/reboot
        file=rm             source=/bin/rm
        file=rmdir          source=/bin/rmdir
        file=sh             source=/bin/sh
        file=sync           source=/bin/sync
        file=touch          source=/bin/touch
        file=true           source=/bin/true
        file=tset           source=/bin/tset
        file=uname          source=/bin/uname
directory=/etc use=/etc
directory=/tmp use=/tmp
directory=/mnt
```

SEE ALSO

Utility Programs - `mkimage()`

NAME

`netrc` – file for `ftp` remote login data

DESCRIPTION

The `.netrc` file contains data for logging into a remote host over the network for file transfers by `ftp`. This file resides in the user's home directory on the machine initiating the file transfer. Its permissions should be set to disallow read access by group and others (see `chmod`).

`account` *string*
: An additional account password. If *string* is present, autologin supplies the specified string if the remote server requires an additional account password; autologin initiates an `ACCT` command if it does not.

`login` *name*
: A user on a remote machine. If *name* is present, autologin initiates a login using the specified name.

`macdef` *name*
: A macro. *name* functions as the `ftp` `macdef` command function. A macro is defined with the specified name; its contents begin with the next `.netrc` line and continue until a null line (consecutive `NEWLINE` characters) is encountered. If `init` is defined, it is automatically executed as the last step in the autologin process.

`machine` *name*
: A remote machine name. The autologin process searches the `.netrc` file for a matching remote machine name, specified on the `ftp` command line (or as an open command argument). Once a match is made, subsequent `.netrc` tokens are processed, stopping at `EOF` or when another machine *name* is encountered.

`password` *string*
: A password. If *string* is present, autologin supplies the specified string if the remote server requires a password as part of the login process. If *string* appears in `.netrc`, `ftp` aborts autologin if the `.netrc` is readable by anyone besides the user.

EXAMPLE

The command, `machine` *merlin* `login` *test* `password` *let_me_in* allows an autologin to the machine *merlin* using the login name *test* with password *let_me_in*.

FILES

`.netrc`

SEE ALSO

Utility Programs - `chmod`, `ftp`, `ftpd`

NAME

networks – network name data base

DESCRIPTION

The networks file contains information regarding the known networks which comprise the DARPA Internet. For each network a single line should be present with the following information:

official-network-name network-number aliases

Items are separated by any number of blanks and/or tab characters. A "#" indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file. This file is normally created from the official network data base maintained at the Network Information Control Center (NIC), though local changes may be required to bring it up to date regarding unofficial aliases and/or unknown networks.

network-number may be specified in the conventional "." notation using the inet_network routine from the Internet address manipulation library. network names may contain any printable character other than a field delimiter, newline, or comment character.

NOTE

A name server should be used instead of a static file. A binary indexed file format should be available for fast access.

FILES

/etc/networks

SEE ALSO

Library Calls - getnetent()

NAME

`nodetab` – special file table

SYNOPSIS

`/etc/nodetab`

DESCRIPTION

The **`/etc/nodetab`** file contains names and descriptions of special files (nodes). Each line is of the form

name:*type*:*major*:*minor*:*perm*

name is a file name for the node to be made. *type* is one of `b`, `c`, `f`, or `p`, for block, character, or FIFO files, or name to create a symbolic link to device `name`. *major* and *minor* give the major and minor device numbers for the node; these fields are not important for FIFOs.

perm is an optional octal number which will be used to set the permissions for the node. If `perm` is not given, the default, 0666, will be used. The value of `perm` will be modified by the `umask` environment variable, so the actual permissions given to the `node` may be different from those in the `nodetab` file.

The `nodetab` file is used by `mknod` when it is given the **`-a`** switch. The file is made much more useful by omitting all but the base name of the special files; for example, `sd1` instead of `/dev/sd1`. If such relative rather than absolute pathnames are used, the same `nodetab` file can be used to create the standard complement of device files in other directories, such as the `/dev` directory of a newly-created file system.

The `nodetab` file can be created by hand, but is more often produced during system configuration by `config`.

SEE ALSO

Utility Programs - `mknod, reboot`

System Calls - `mknod(), umask()`

NAME

`param, kparam` – LynxOS kernel invariants and variables

SYNOPSIS

```
#include <sys/param.h>
#include "kparam.h"
```

DESCRIPTION

The configuration of a LynxOS kernel is determined, in part, by the two headers, `<sys/param.h>` and `kparam.h`. `<sys/param.h>` is found in the include directory hierarchy, while `kparam.h` is found in the build directory for individual kernel configurations (in other words, `/sys/BANANA/kparam.h` for the kernel configured using the `BANANA` configuration file).

`<sys/param.h>` contains invariant information which you cannot alter. This stuff is hardwired into the LynxOS implementation and is not configurable without source (and in many cases, it's not that easy even *with* source!)

`kparam.h` contains details which can be changed for each new configuration of LynxOS. `kparam.h` includes kernel numeric parameters, #defines that reflect the configurable modules in the system, and information about the device driver configuration of the configured kernel.

NOTE

`kparam.h` is a new header file. Previously, all configuration information existed in `<sys/param.h>`. The split is necessary to support multiple kernel configurations on one system. If you need information for your application which was previously in `<sys/param.h>`, and which is now in `kparam.h`, it means that this information is variable depending on which kernel is running. You should consider retrieving this information dynamically from the running kernel using `info` or `sysconf`.

SEE ALSO

Utility Programs – `kernconfig`

Library Functions - `info(), sysconf()`

File Formats – `kernconfig, cfg`

NAME

`passwd` – table of user names and numbers

SYNOPSIS

/etc/passwd
#include <pwd.h>

DESCRIPTION

The `/etc/passwd` file contains lines giving names and numbers for all users known to the system. Each line is of the form:

```
name : password : uid : gid : info : dir : shell
```

`name` is a string used to identify the user with ID number `uid` in group number `gid`. The `info` field may contain any information at all. `dir` is the home directory of the user, and `shell` is the pathname of the user's login shell. The `password` field contains a printable but unintelligible form of the user's password. If the field is empty, no password will be required of the user by `login`. If set to an asterisk, `login` will never allow access to the system via the user name in that entry.

The `getpwent` routines retrieve information from `/etc/passwd` and return it in the form of `passwd` structures, defined in `pwd.h`

```
struct passwd {
  char  *pw_name;    /* User name */
  char  *pw_passwd;  /* Password (encrypted) */
  int  pw_uid;       /* Used ID number */
  int  pw_gid;       /* Group ID number */
  int  pw_quota;     /* Always zero; not in passwd file */
  char  *pw_comment; /* Information field */
  char  *pw_gecos;   /* Unused, always empty */
  char  *pw_dir;     /* Home directory */
  char  *pw_shell;   /* Login shell */
};
```

The password file is used by `login` and by various other system programs which must translate user ID numbers into user names, user names to directory names, or what have you.

SEE ALSO

Utility Programs - `login, passwd`

Library Calls - `getpwent(), crypt()`

NAME

printcap – printer capability database

SYNOPSIS

/etc/printcap

DESCRIPTION

printcap is a simplified version of the termcap data base used to describe line printers. The spooling system accesses the printcap file every time it is used, allowing dynamic addition and deletion of printers. Each entry in the data base is used to describe one printer. This data base may not be substituted for, as is possible for termcap, because it may allow accounting to be bypassed.

The default printer is normally lp, though the environment variable PRINTER may be used to override this. Each spooling utility supports an option, **-P** printer, to allow explicit naming of a destination printer.

CAPABILITIES

Refer to termcap for a description of the file layout. The printer configuration options are shown in the table “Printer Configuration Options”.

Table 1: Printer Configuration Options

Field Code	Type	Default	Description
af	str	NULL	Name of accounting file
br	num	none	If lp is a tty, set the baud rate (ioctl call)
cf	str	NULL	cifplot data filter
df	str	NULL	Text data filter (DVI format)
fc	num	0	If lp is a tty, clear flag bits (sgtty.h)
ff	str	f	String to send for a form feed
fo	bool	false	Print a form feed when device is opened

Table 1: Printer Configuration Options

Field Code	Type	Default	Description
fs	num	0	Like fc but set bits
gf	str	NULL	Graph data filter (plot (3X) format)
hl	bool	false	Print the burst header page last
ic	bool	false	Driver support (non-standard) ioctl to indent printout
if	str	NULL	Name of text filter which does accounting
lf	str	/dev/console	Error logging filename
lo	str	lock	Name of lock file
lp	str	/dev/lp	Device name to open for output
mx	num	1000	Maximum file size (in BUFSIZ blocks), zero = unlimited
nd	str	NULL	Next directory for list of queues (unimplemented)
nf	str	NULL	ditroff data filter (device independent troff)
of	str	NULL	Name of output filtering program
pc	num	200	Price per foot or page in hundredths of cents
pl	num	66	Page length (in lines)
pw	num	132	Page width (in characters)
px	num	0	Page width in pixels (horizontal)
py	num	0	Page length in pixels (vertical)
rf	str	NULL	Filter for printing FORTRAN style text files
rg	str	NULL	Restricted group. Only members of group allowed access

Table 1: Printer Configuration Options

Field Code	Type	Default	Description
`rm`	`str`	`NULL`	Machine name for remote printer
`rp`	`str`	`lp`	Remote printer name argument
`rs`	`bool`	`false`	Restrict remote users to those with local accounts
`rw`	`bool`	`false`	Open the printer device for reading and writing
`sb`	`bool`	`false`	Short banner (one line only)
`sc`	`bool`	`false`	Suppress multiple copies
`sd`	`str`	`/usr/spool/lpd`	Spool directory
`sf`	`bool`	`false`	Suppress form feeds
`sh`	`bool`	`false`	Suppress printing of burst page headers
`st`	`str`	`status`	Status file name
`tf`	`str`	`NULL`	`troff` data filter (`cat phototypesetter`)
`tr`	`str`	`NULL`	Trailer string to print when queue empties
`vf`	`str`	`NULL`	Raster image filter
`xc`	`num`	`0`	If `lp` is a `tty`, clear local mode bits (`tty(4)`)
`xs`	`num`	`0`	Like `xc` but set bits

If the local line printer driver supports indentation, the daemon must understand how to invoke it.

FILTERS

The `lpd` daemon creates a pipeline of filters to process files for various printer types. The filters selected depend on the flags passed to `lpr`, as shown in the table "lpr Flag To printcap Field Code Mapping".:

Table 2: `lpr` **Flag To** `printcap` **Field Code Mapping**

`lpr` **Flag**	`printcap` **Field Code**	**Filter Type**
-p	`pr` | `if`	Regular text + `pr`
none	`if`	Regular text
-c	`cf`	`cifplot`
-d	`df`	DVI (text)
-g	`gf`	Plot
-n	`nf`	`ditroff`
-f	`rf`	Fortran
-t	`tf`	`troff`
-v	`vf`	Raster image

The command line received by the input filter is in the following format:

if `[`**-c**`]` **-w**`width` **-l**`length` **-i**`indent` **-n** `login` **-h** `host acct-file`

The **-c** flag is passed only if the **-l** flag (pass control characters literally) is specified to `lpr`. Width and length specify the page width and length (from `pw` and `pl` respectively) in characters. The **-n** and **-h** parameters specify the login name and host name of the owner of the job respectively. `acct-file` is passed from the `af` `printcap` entry.

If no value is specified for `if`, the value of `of` is used instead, with the distinction that `of` is opened only once, while *if* is opened for every individual job. Thus, the input filter is better suited to performing accounting. The `output filter` is only given the `width` and `length` flags.

All other filters are called as:

filter **-x**width **-y**length **-n** login **-h** host acct-file

Where `width` and `length` are represented in pixels, specified by the `px` and `py` entries respectively.

All filters take `stdin` as the file, `stdout` as the printer, may log either to `stderr` or using `syslog`, and must not ignore `SIGINT`.

LOGGING

Error messages generated by the line printer programs themselves (that is, the `lp*` programs) are logged by `syslog` using the `LPR` facility. Messages printed on stderr of one of the filters are sent to the corresponding `lf` file. The filters may, of course, use `syslog` themselves.

Error messages sent to the console have a carriage return and a line feed appended to them, rather than just a line feed.

SEE ALSO

Utility Programs -	`lpc, lpd, pac, lpr, lpq, lprm`
File Formats -	`termcap`
Documents -	*4.3BSD Line Printer Spooler Manual*

NAME

`protocols` – protocol name data base

DESCRIPTION

The `protocols` file contains information regarding the known protocols used in the DARPA Internet. For each protocol a single line should be present with the following information:

official-protocol-name protocol-number aliases

Items are separated by any number of blanks and/or tab characters. A “`#`” indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file.

Protocol names may contain any printable character other than a field delimiter, newline, or comment character.

NOTE

A name server should be used instead of a static file. A binary indexed file format should be available for fast access.

FILES

`/etc/protocols`

SEE ALSO

Library Functions - `getprotoent()`

NAME

`rcsfile` – format of `RCS` file

DESCRIPTION

An `rcsfile` is an ASCII file. Its contents are described by the grammar below. The text is free format; i.e., spaces, tabs and new lines have no significance except in strings. Strings are enclosed by “`@`”. If a string contains a “`@`”, it must be doubled.

The meta syntax uses the following conventions: “|” (bar) separates alternatives; “{” and “`}`” enclose optional phrases; “`{`” and “`}*`” enclose phrases that may be repeated zero or more times; “`{`” and “`}+`” enclose phrases that must appear at least once and may be repeated; “<” and “>”enclose nonterminals.

```
<rcstext>  ::=  <admin> {<delta>}* <desc> {<deltatext>}*
<admin>  ::=    head   {<num>};
    access   {<id>}*;
    symbols   {<id> : <num>}*;
    locks   {<id> : <num>}*;
    comment   {<string>};
<delta>  ::=  <num>
    date   <num>;
    author   <id>;
    state   {<id>};
    branches   {<num>}*;
    next   {<num>};
<desc>   ::=   desc   <string>
<deltatext>  ::=   <num>
    log   <string>
    text   <string>
<num>  ::=  {<digit>{.}}+
<digit>  ::=   0 | 1 | ... | 9
<id>  ::=  <letter>{<idchar>}*
<letter>  ::=   A | B | ... | Z | a | b | ... | z
<idchar>  ::=   Any printing ASCII character except space,
    tab, carriage return, new line,and
    <special>.
<special>  ::=   ; | : | , | @
<string>  ::=  @{any ASCII character, with '@' doubled}*@
```

Identifiers are case sensitive. Keywords are in lower case only. The sets of keywords and identifiers may overlap.

The `<delta>` nodes form a tree. All nodes whose numbers consist of a single pair (e.g., 2.3, 2.1, 1.3, etc.) are on the "trunk", and are linked through the "next" field in order of decreasing numbers. The "head" field in the `<admin>` node points to the head of that sequence (i.e., contains the highest pair).

All `<delta>` nodes whose numbers consist of `2n` fields `(n>=2)` (e.g., 3.1.1.1, 2.1.2.2, etc.) are linked as follows. All nodes whose first `(2n)-1` number fields are identical are linked through the "next" field in order of increasing numbers. For each such sequence, the `<delta>` node whose number is identical to the first `2(n-1)` number fields of the deltas on that sequence is called the branchpoint. The "branches" field of a node contains a list of the numbers of the first nodes of all sequences for which it is a branchpoint. This list is ordered in increasing numbers.

COPYRIGHT NOTICE

The RCS system is the work of Walter F. Tichy of Purdue University. It was obtained by Lynx from the Free Software Foundation. As such, Lynx Real-Time Systems will make available for nominal postage and materials cost complete source code for all parts of the RCS system. Furthermore, LynxOS users are hereby permitted to copy and distribute RCS at will under these same terms; that is, notice that source is available from Lynx Real-Time Systems must be given along with any RCS distribution, or else the source itself. If any RCS source is to be used as a component of some other software system, the Free Software Foundation should be contacted for details:

Free Software Foundation, Inc.
675 Massachusetts Ave.
Cambridge, MA 02139

Note that RCS as distributed by FSF has been modified in small ways by Lynx Real-Time Systems, Inc., and thus although source for RCS may be available elsewhere it will be <u>different</u> from LynxOS RCS. Lynx Real-Time Systems assumes no responsibility for copies of RCS obtained elsewhere.

SEE ALSO

Utility Programs - `co, ident, rcs, rcsdiff, rcsintro, rlog`

File Formats - `rcsfile`

NAME

rhosts – user-specified file of equivalent hosts and users

DESCRIPTION

This file resides in a user's login directory. It contains entries, one per line, which are of the form:

hostname username

It allows a user to specify a set of users of other systems which are allowed equivalent capabilities to himself on this system. The *username* field is optional.

In an environment where a single organization might have many systems used by a common set of users, it is often the case that a single user will have login IDs on many different systems. In the common case where the login IDs are the same for each user on all systems, then user authentication is provided by the list of host names in `/etc/hosts.equiv`. In the case where a host is not in `/etc/hosts.equiv`, or the user has a different name on another system, the user can provide individual authentication by adding entries in his personal `.rhosts` file. Users who connect to the system, via `rcp`, `rsh`, or `rlogin` and are authorized via `.rhosts`, will have privileges on this system exactly equivalent to the user granting authorization.

`rshd`, used to support `rsh` and `rcp` requests, uses `.rhosts` in the following way. When the connection is made, `rshd` gets the ID of the user on the remote ("calling") system. It then looks up the remote user in the local `/etc/passwd` file. If the remote user is not the super-user, then `/etc/hosts.equiv` is checked for the name of the remote host. If it is found, the user is considered to be equivalent to the user of the same local name, and the command proceeds. If the host name is not found, or if the remote user is the super-user, then `rshd` checks the file `.rhosts` in the login directory found in `/etc/passwd`. If an entry is found for the remote host, or this local user name and remote host combination, then the user is considered equivalent and the command proceeds. If this test fails, the command is terminated. `rlogin` uses these files in an analogous fashion.

The hostname entry specified in `.rhosts` must be the "primary" host name, i.e. the first one in `/etc/hosts`.

FILES

`$HOME/.rhosts`

SEE ALSO

Utility Programs - `rcp, rlogin, rsh, rlogind, rshd`

File Formats - `hosts.equiv`

NAME

`rpc` – RPC program number data base

SYNOPSIS

`/etc/rpc`

DESCRIPTION

The RPC file contains user readable names that can be used in place of `rpc` program numbers. Each line has the following information:

name of server for the RPC *program*
RPC *program number*
aliases

Items are separated by any number of blanks and/or tab characters. A "#" indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file.

Here is an example of the `/etc/rpc` file from the Sun RPC Source distribution.

```
#
# rpc 88/08/01 4.0 RPCSRC; from 1.12 88/02/07 SMI
#
portmapper   100000   portmap sunrpc
rstatd       100001   rstat rsta_svc rup perfmeter
rusersd      100002   rusers
nfs          100003   nfsprog
ypserv       100004   ypprog
mountd       100005   mount showmount
ypbind       100007
walld        100008   rwall shutdown
yppasswdd    100009   yppasswd
etherstatd   100010   etherstat
rquotad      100011   rquotaprog quota rquota
sprayd       100012   spray
3270_mapper  100013
rje_mapper   100014
selection_svc 100015  selnsvc
database_svc 100016
```

```
rexd         100017   rex
alis         100018
sched        100019
llockmgr     100020
nlockmgr     100021
x25.inr      100022
statmon      100023
status       100024
bootparam    100026
ypupdated    100028   ypupdate
keyserv      100029   keyserver
tfsd         100037
nsed         100038
nsemntd      100039
```

FILES

`/etc/rpc`

SEE ALSO

Library Functions - `getrpcent()`

NAME

`services` – service name data base

DESCRIPTION

The services file contains information regarding the known services available in the DARPA Internet. For each service a single line should be present with the following information:

```
official-service-name  port-number/protocol-name  aliases
```

Items are separated by any number of blanks and/or tab characters. The port number and protocol name are considered a single item; a "/" is used to separate the port and protocol (e.g. "512/tcp"). A "#" indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file.

Service names may contain any printable character other than a field delimiter, newline, or comment character.

NOTE

A name server should be used instead of a static file. A binary indexed file format should be available for fast access.

FILES

`/etc/services`

SEE ALSO

Library Functions - `getservent()`

NAME

`sm, state - rpc.statd` directory and file structures

SYNOPSIS

/etc/state

/etc/sm

DESCRIPTION

The file `/etc/state` and the directory `/etc/sm` are maintained by the `rpc.statd` daemon. Each entry in the `/etc/sm` directory contains the host names of the nodes that the `rpc.statd` monitors. These entries will be added and deleted, depending up on any pending NFS file locking issues with those hosts. When the system is rebooted, the `rpc.statd` looks into this directory to notify the nodes of the new state of the self host. The `/etc/state` contains the new state information of the host. This will be incremented each time the host gets rebooted.

FILES

`/etc/sm`

`/etc/state`

SEE ALSO

Utility programs - `rpc.statd`, `rpc.lockd.svc`, `rpc.lockd.clnt`

NAME

stab – symbol table entries

SYNOPSIS

#include <stab.h>

DESCRIPTION

stab entries in an a.out file symbol table are those with one of the high three bits set in the n_type field. These entries are generated by the compiler and passed along by the linker into the final executable file.

The defined values for the n_type field for stab entries are listed below. The comments give the values of the fields in order: n_name, n_type, n_other, n_desc, and n_value.

```
#define N_GSYM 0x20  /* global symbol */
#define N_FUN 0x24   /* procedure */
#define N_LCSYM 0x28 /* .lcomm symbol */
#define N_RSYM 0x40  /* register sym */
#define N_SSYM 0x60  /* structure elt */
#define N_LSYM 0x80  /* local sym */
#define N_PSYM 0xa0  /* parameter */
#define N_LBRAC 0xc0 /* left bracket */
#define N_EFLDS 0xc4 /* end fields */
#define N_BCOMM 0xe2 /* begin common */
#define N_ECOML 0xe8 /* end common (local name) */
#define N_FLIST 0xfc /* 3rd stab w/ index of struct field info*/
#define N_FNAME 0x22 /* procedure name */
#define N_STSYM 0x26 /* static symbol */
#define N_PC 0x30    /* global pascal symbol */
#define N_SLINE 0x44 /* src line */
#define N_SO 0x64    /* source file name */
#define N_SOL 0x84   /* #included file name */
#define N_ENTRY 0xa4 /* alternate entry */
#define N_BFLDS 0xc2 /* begin fields */
#define N_RBRAC 0xe0 /* right bracket */
#define N_ECOMM 0xe4 /* end common */
#define N_LENG 0xfe  /* 2nd stab entry with length information */
```

When the `n_desc` symbol table entry field denotes a type, it takes the form of a `desc` structure, described below.

```
struct desc {
  unsigned int q6:2, q5:2, q4:2, q3:2, q2:2, q1:2, basic:4;
};
```

The fields `q1` through `q6` qualify the basic type described by basic. The possible values for the qualifications fields are:

0	none
1	pointer
2	function
3	array

The basic type field may be any of these values:

0	undefined
1	function argument
2	character
3	short
4	int
5	long
6	float
7	double
8	structure
9	union
10	enumeration
11	member of enumeration
12	unsigned character
13	unsigned short

14	unsigned int
15	unsigned long

The subtype field of the N_PC entry is used to do type checking across separately compiled files. Its intent is to be used with Pascal. Currently no language processor produces these entries. The list of values for the subtype field is given below.

1	source file name
2	included file name
3	global label
4	global constant
5	global type
6	global variable
7	global function
8	global procedure
9	external function
10	external procedure
11	library variable
12	library routine

SEE ALSO

Utility Programs - `nm`

Library Functions - `nlist()`

File Formats - `a.out`

NAME

`starttab` – startup data file

SYNOPSIS

`/etc/starttab`

DESCRIPTION

The **`/etc/starttab`** file contains information used by `init` to conduct the boot process and to configure multi-user tasks. A line of the startup file may start with a pound sign (#) and contain a comment. Otherwise, the line should be one of the following pieces of information:

Field	Meaning	Usual Value
`umask`	Default umask value, in octal	`022`
`shell`	Name of single user shell program	`/bin/dlsh`
`rc`	Name of `rc` program	`/bin/rc`
`data`	Data segment limit, in KB	`512`
`stack`	Stack segment limit, in KB	`64`
`core`	Core file limit, in KB	`512`

The data above must appear in the order given. The file can be truncated at any point, in which case default values will be used for the remainder. If there isn't any `rc` file the, Usual Value field should have `NOTHING` in it.

SEE ALSO

Utility Programs - `init`

NAME

`superblock` – file system master entry

SYNOPSIS

`#include <disk.h>`

DESCRIPTION

The superblock contains information locating and describing vital components of the file system on a device. The superblock is always block number one. Its description is found in **`disk.h`**.

```
struct superblock {
  off_t free_inodes; /* free inode list */
  off_t free_blocks; /* free_block list */
  long  num_inodes;  /* number of inodes on this device */
  long  num_blocks;  /* number of blocks on this device */
  long  root_inode;  /* root of disk file system */
  off_t bad_blocks;
  long  free_icount; /* number of free inodes */
  long  free_bcount; /* number of free blocks */
  long  code_name;   /* name of this disk */
  long  status;      /* dirty or ok */
  long  magic;       /* magic number */
  long  time;        /* creation time */
};
```

The `free_inodes` and `free_blocks` fields point to the lists, which are kept on disk, of free inodes and free disk blocks. The `num_inodes` and `num_blocks` fields are set at file system creation time to appropriate permanent values. The `root_inode` field contains the number of the root inode, which is always equal to 1 (this field is used as a sanity check for the file system). The `free_icount` and `free_bcount` fields record the current number of free inodes and disk blocks, and are used mostly by the **`df`** utility program. The `magic` field is always equal to `SB_MAGIC`, as defined in **`disk.h`**. The `time` field is set by `mkfs` when the file system is created. Note that the `magic` and `time` fields are specified only on newer file systems, which pack four inodes per block, and not on older file systems, which pack eight inodes per block. Currently, LynxOS does not maintain its own bad block list; this is normally done by the controller hardware of hard disk drives.

The superblock is created by `mkfs` and maintained by the operating system; it is only available for inspection, not modification.

SEE ALSO

Utility Programs - `df, fsck, mkfs`

File Formats - `inode`

NAME

`syslog.conf` – configuration file for `syslogd`

DESCRIPTION

The `syslog.conf` file is the configuration file for the `syslogd(1)` program. It consists of lines with two fields: the selector field which specifies the types of messages and priorities to which the line applies, and an action field which specifies the action to be taken if a message syslogd receives matches the selection criteria. The selector field is separated from the action field by one or more tab characters.

The Selectors function are encoded as a facility, a period (.), and a level, with no intervening white-space. Both the facility and the level are case insensitive.

The facility describes the part of the system generating the message, and is one of the following keywords: `auth`, `authpriv`, `cron`, `daemon`, `kern`, `lpr`, `mail`, `mark`, `news`, `syslog`, `user`, `uucp` and `local0` through `local7`. These keywords (with the exception of mark) correspond to the similar `LOG_` values specified to the `openlog(3)` and `syslog(3)` library routines.

The level describes the severity of the message, and is a keyword from the following ordered list (higher to lower): `emerg`, `alert`, `crit`, `err`, `warning`, `notice` and `debug`. These keywords correspond to the similar `LOG_` values specified to the `syslog` library routine.

See `syslog(3)` for a further descriptions of both the facility and level keywords and their significance.

If a received message matches the specified facility and is of the specified level (or a higher level), the action specified in the action field will be taken.

Multiple selectors may be specified for a single action by separating them with semicolon (;) characters. It is important to note, however, that each selector can modify the ones preceding it.

Multiple facilities may be specified for a single level by separating them with comma (,) characters.

An asterisk (*) can be used to specify all facilities or all levels.

The special facility “mark” receives a message at priority “info” every 20 minutes (see `syslogd(1)`). This is not enabled by a facility field containing an asterisk.

The special level “none” disables a particular facility.

The action field of each line specifies the action to be taken when the selector field selects a message. There are four forms:

- A pathname (beginning with a leading slash). Selected messages are appended to the file.
- A hostname (preceded by an at (@) sign). Selected messages are forwarded to the syslogd program on the named host.
- A comma separated list of users. Selected messages are written to those users if they are logged in.
- An asterisk. Selected messages are written to all logged-in users.

Blank lines and lines whose first non-blank character is a hash (#) character are ignored.

EXAMPLES

A configuration file might appear as follows:

```
# Log all kernel messages, authentication messages of
# level notice or higher and anything of level err or
# higher to the console.
# Don't log private authentication messages!
*.err;kern.*;auth.notice;authpriv.none   /dev/con

# Log anything (except mail) of level info or higher.
# Don't log private authentication messages!
*.info;mail.none;authpriv.none /etc/log/messages

# The authpriv file has restricted access.
authpriv.* /etc/log/secure

# Log all the mail messages in one place.
mail.* /etc/log/maillog

# Everybody gets emergency messages, plus log them on another
# machine.
*.emerg *
*.emerg @arpa.berkeley.edu

# Root and Eric get alert and higher messages.
*.alert root,eric

# Save mail and news errors of level err and higher in a
# special file.
uucp,news.crit /etc/log/spoolerr
```

FILES

`/etc/syslog.conf`	The `syslogd(1)` configuration file.

BUGS

The effects of multiple selectors are sometimes not intuitive. For example:

```
mail.crit,*.err
```

will select `mail` facility messages at the level of `err` or higher, not at the level of `crit` or higher.

SEE ALSO

Utility Programs - `syslogd`

Library Functions - `syslog()`

HISTORY

The `syslog.conf` file format is currently under development.

COPYRIGHT

NAME

`tar` – tape archive file format

SYNOPSIS

`#include <tar.h>`

DESCRIPTION

The `tar` utility copies multiple files into a single archive which can be easily placed on a tape, floppy, etc.

Each file that is stored in the `tar` archive is split into zero or more blocks (of size `TBLOCK`), and is preceded by a header block, which contains various information about the file. Immediately following the last file data block are two blocks of binary zeros, which indicate the end of the archive.

Since `tar` is frequently used to write files to a tape or floppy, it groups individual blocks into groups of *n* blocks (the default is 20) for better I/O performance. This grouping can be changed by using the `-b` option of `tar`. If the last file does not completely fill the *n* blocks, `tar` will pad to the appropriate size after writing the two blocks of zeros. The padded blocks contain random data. When reading the `tar` archive, `tar` will automatically adjust the group size.

The format of the header block is as follows:

```
#define  TBLOCK  512
#define  NAMSIZ  100
#define  TMODLEN  8
#define  TUIDLEN  8
#define  TGIDLEN  8
#define  TSIZLEN  12
#define  TMTMLEN  12
#define  TCKSLEN  8
#define  TMAGLEN  6
#define  TVERSLEN 2
#define  TUNMLEN  32
```

```
#define  TGNMLEN  32
#define  TDEVLEN  8
#define  PFXSIZ  155
union hblock {
  char dummy[TBLOCK];
  struct header {
    char  name[NAMSIZ];
    char  mode[TMODLEN];
    char  uid[TUIDLEN];
    char  gid[TGIDLEN];
    char  size[TSIZLEN];
    char  mtime[TMTMLEN];
    char  chksum[TCKSLEN];
    char  typeflag;
    char  linkname[NAMSIZ];
    /* Above is BSD-compatible; below is from POSIX */
    char  magic[TMAGLEN];
    char  version[TVERSLEN];
    char  uname[TUNMLEN];
    char  gname[TGNMLEN];
    char  devmajor[TDEVLEN];
    char  devminor[TDEVLEN];
    char  prefix[PFXSIZ];
  } dbuf;
};
```

The `name` and `linkname` fields are null-terminated strings. The `mode`, `uid`, `gid`, `size`, `mtime`, and `chksum` fields are leading zero-filled octal numbers in ASCII. Each of these octal numbers is followed by a space, and is null terminated, except for `size` and `mtime`, which are not null terminated.

The `tar` header block limits the filename or `linkname` of each file in the archive to `NAMSIZ` bytes. In this case, `name/linkname` consists of the entire path of an individual file/link, up to `NAMSIZ-1` bytes. `mode` is the file mode, with the high four bits masked off. `uid` and `gid` are the user and group numbers of the owner of the file. `size` is the size of the file in bytes. Hard links and symbolic links have a size of zero. `mtime` is the modification time of the file at the time it was added to the archive. `chksum` is the sum of all the bytes in the header block. The `chksum` field is treated as if it were all blanks, when calculating the checksum. `linkflag` is an ASCII character which identifies the type of the archived file, the values used are defined in `tar.h`. Any field not used for a particular file will contain binary zeros (and will be included in the checksum).

The POSIX extensions to the `tar` header block have added seven new fields, and have slightly modified others. The `name`, `linkname`, and `prefix` fields are normally null terminated strings, although null termination is not required in the case where all `PFXSIZ` or `NAMSIZ` bytes are used to store the `prefix` or `name/linkname`. This allows the maximum path to be `PFXSIZ+NAMSIZ+1` bytes, rather than just `NAMSIZ` bytes allowed without the POSIX extensions.

The reason that the maximum path size is larger than the arrays used to store it, is that the `'/'` character which separates the `prefix` from the `name/linkname` is not actually stored in either of the arrays. The `magic`, `uname`, and `gname` fields are null-terminated strings. The `magic` field contains the string `"ustar"`. `uname`, and `gname` contain the ASCII strings, truncated if necessary, that correspond to the owner and group of the file. The `version` field contains the string `"00"` and is not null terminated. The `devmajor` and `devminor` fields are octal strings similar to the octal strings described above.

Both the `tar` header and the POSIX extensions have been designed to be portable across machines.

SEE ALSO

Utility Programs - `tar`

NAME

tconfig – serial port configuration file

SYNOPSIS

/etc/tconfig

DESCRIPTION

The file **/etc/tconfig** contains descriptions of serial port configurations. A configuration is simply a state of all the various tty driver options. For example, a configuration can dictate whether typed characters are to be echoed, which control key should cause a SIGINT signal, or the baud rate of the port.

CONFIGURATION DESCRIPTIONS

The tconfig file consists of comments on lines beginning with a pound sign (#), and description entries. An entry in the tconfig file begins with an identifier for the configuration being described. Following this is a colon-separated list of descriptors, which define strings, numeric parameters, and the state of various boolean flags. All descriptors have a two character name, traditionally but not necessarily alphanumeric. The name is followed by an equals sign (=) if it is a string capability, a pound sign(#) if it is a numeric parameter, or nothing (except the field-terminating colon) for boolean fields. For the configuration options recognized by ttyconfig, see the table "Configuration Options Recognized By ttyconfig".

Table 1: Configuration Options Recognized By ttyconfig

Descriptor	Type	Field	Meaning
bk	str	t_brkc	Line break character
ds	str	t_dsuspc	Suspend (sends SIGTSTP when read)
e2	str	sg_2erase	Second erase character
ef	str	t_eofc	End-of-file character
er	str	sg_erase	Erase character
fl	num	sg_flags	Primary flag word

Table 1: Configuration Options Recognized By `ttyconfig`

Descriptor	Type	Field	Meaning
`fs`	`str`	`t_flushc`	Toggle output flushing
`in`	`str`	`t_intrc`	Interrupt character (sends `SIGINT`)
`kl`	`str`	`sg_kill`	Line erase character
`lm`	`num`	`l_localmode`	Local mode word (more flags)
`ln`	`str`	`t_lnextc`	Literal escape
`qc`	`str`	`t_quitc`	Quit character (sends `SIGQUIT`)
`rp`	`str`	`t_rprntc`	Reprint input line
`si`	`str`	`sg_ispeed`	Input baud rate
`so`	`str`	`sg_ospeed`	Output baud rate
`sp`	`str`	`t_stopc`	Suspend output
`ss`	`str`	`t_suspc`	Suspend character (sends SIGTSTP)
`st`	`str`	`t_startc`	Restart output
`tc`	`str`		Description chaining
`we`	`str`	`t_werasc`	Word erase

Making Configuration Descriptions

Configuration descriptions begin with a list of names for the configuration, separated by vertical bar characters. Usually, a compact form and a long form of the name are given:

```
default|default configuration:
noech|default w/o echo:
raw|default, raw mode:
```

The name will be used by the `ttyconfig` routine when a program requests a reconfiguration. The name is also recorded in the `/etc/ttys` file to identify the default configuration for some serial port.

Following the name is a colon-separated list of descriptors. Usually, the list is quite long, so the description may be broken (between descriptors) by ending a line with a backslash (\) and continuing on the next line. The first non-whitespace character on the next line must be a colon, introducing a new descriptor.

Each descriptor field gives a value for one of the fields listed in the table. Meaningful values for the fields are defined in the `tty` driver documentation. Only the first character of a character definition field is significant. Fields which are not included in a definition will not be changed by a call to `ttyconfig` with that definition; thus a definition can be used to modify the current state without resetting all characteristics.

Within string descriptors, the special sequences from the table “String Descriptor Special Sequences” are recognized.

Table 2: String Descriptor Special Sequences

Special Sequence	Meaning
`\E`	The ASCII escape character, control-`[`.
`\n`	Newline (linefeed).
`\r`	Carriage return.
`\f`	Form feed.
`\t`	Tab.
`\b`	Backspace.
`^X`	Control-X, for any printable character X .
`\nnn`	Character corresponding to the octal number`nnn` .
`\^`	The uparrow (caret) character.
`\\`	The backslash character.

If a colon must be included in a string, it must be written as `\072`. Null characters should be entered as `\200`.

Baud rates, though in truth numeric, are defined as strings. Abbreviations for baud rates can be used, for example “96” for “9600”. Most often, the input and output speeds are set to the same value.

Flag word definitions are somewhat difficult to create. The `stty` program prints the values of the flag words in octal in response to the `all` switch, so values for the `tconfig` file can be derived by setting a terminal with `stty` and then looking at the numeric value.

The Default Description

The `tconfig` file should always contain at least one description, usually named “default”. This description should contain standard control character definitions, and definitions for the flag words. An example default definition is:

```
df|default|default setup: \
    :si=19200:so=19200: \
    :er=^H:e2=\177:we=^W:kl=^U: \
    :fl#0000030: \
    :in=^C:qc=034: \
    :st=^Q:sp=^S: \
    :ef=^D: \
    :bk=: \
    :lm#0012005: \
    :ss=^Z:ds=^Y: \
    :rp=^R:fs=^O: \
    :ln=^V:
```

In this description, the speed is set to 19.2 kilobaud. Two erase characters, backspace (`^H`) and delete, are defined and will operate identically. Word erase is set to `^W`, and line erase is set to `^U`. The interrupt (`SIGINT`) character is set to the common `^C`, and the quit (`SIGQUIT`) character is set to `^\`. Output suspend/resume is set to `^S/^Q`, and `^D` is used as end-of-file marker. The extra line break character is left unset. The two stop characters (`SIGTSTP`) are set to `^Z` and `^Y`, the most familiar values. Line reprint is `^R`. Output flushing is toggled with `^O`. Special characters can be entered untouched by the driver by prefixing them with the literal escape character `^V`. Finally, the primary flag word is set to `CRMOD` and `ECHO`, and the local flag word is set to `LCTLECH`, `LCRTKIL`, `LCRTERA`, `LCRTBS`.

If a description is to be created which is a slight modification of another already in the file, a new description can be made by defining the fields which are different and then referencing the original description with a `tc` entry. For example, a description which is identical to the “default” description above, but without the stop or delayed stop characters, could be created as:

```
nostop:ss=:ds=:tc=default
```

SEE ALSO

Utility Programs -	`stty, tset`
System Calls -	`ioctl()`
Library Functions -	`ttyconfig()`
Device Facilities -	`tty`
File Formats -	`termcap`

NAME

`termcap` – terminal capability descriptions

SYNOPSIS

`/etc/termcap`

DESCRIPTION

The file `/etc/termcap` contains a list of descriptions of particular models of terminals. The file is consulted by programs which wish to use typical terminal features such as cursor positioning or line editing without being tied to a particular type of terminal. New entries can be added for newly-acquired terminals, and new capabilities may be defined for special applications.

CAPABILITIES

The `termcap` file consists of comments, on lines beginning with a pound sign (#), and description entries. An entry in the `termcap` file begins with an identifier for the terminal being described. Following this is a colon-separated list of capability descriptors, which define strings, numeric parameters, and the presence or absence of various features. All capabilities have a two character name, traditionally but not necessarily alphanumeric. The name is followed by an equals sign (=) if it is a string capability, a pound sign(#) if it is a numeric parameter, or nothing (except the field-terminating colon) for boolean fields. For the capabilities recognized by LynxOS, refer to the table “Capabilities Recognized By LynxOS”.

Table 1: Capabilities Recognized By LynxOS

Descriptor	Type	Meaning
`al`	`str`	Insert a blank line a
`am`	`bool`	Autowrap at right margin
`bc`	`str`	Backspace character if not `\^H`
`bl`	`str`	Bell character if not \^G
`cd`	`str`	Clear to end of display

Table 1: Capabilities Recognized By LynxOS

Descriptor	Type	Meaning
`ce`	`str`	Clear to end of line
`cl`	`str`	Home cursor and clear screen
`cm`	`str`	Absolute cursor positioning
`co`	`num`	Columns per line
`cs`	`str`	Set scrolling region (ANSI terminals)
`ct`	`str`	Clear programmable tab settings
`dc`	`str`	Delete current character
`dl`	`str`	Delete current line
`do`	`str`	Down one line
`ei`	`str`	End insert mode
`hc`	`bool`	Hardcopy terminal
`if`	`str`	Initialization file name
`im`	`str`	Start insert mode
`ip`	`str`	Dummy string for insert mode padding
`is`	`str`	Initialization string
`li`	`num`	Number of lines on the screen
`nd`	`str`	Move cursor right (non-destructive)
`pc`	`str`	Pad character if not zero
`pt`	`bool`	Terminal has programmable tabs
`se`	`str`	End standout mode
`sf`	`str`	Scroll forwards (up)
`sg`	`num`	Spaces left by `so` or `\b{se}`
`so`	`str`	start standout mode
`sr`	`str`	reverse scroll (down)
`st`	`str`	set programmable tab stop
`tc`	`str`	description chaining

Table 1: Capabilities Recognized By LynxOS

Descriptor	Type	Meaning
ue	str	end underscore mode
ug	num	spaces left by us or \b{ue}
up	str	move cursor up
us	str	start underscore mode
vb	str	visual bell (such as flash screen)

Creating Terminal Descriptions

The first step in creating a terminal description is the invention of a name for the terminal. Names should be long enough to avoid collision with others in the database, but short enough to avoid user irritation. Usually, a compact form and a long form of the name are given, separated by vertical bar characters:

```
wyse60|wyse model 60, 80*24:
```

The name will be used by the `tgetent` routine when a program requests the description for an entry. The name is also recorded in the `/etc/ttys` file to identify the type of terminal normally connected to some serial port.

Following the name is a colon-separated list of descriptors. Usually, the list is quite long, so the description may be broken (between descriptors) by ending a line with a backslash and continuing on the next line. The first non-whitespace character on the next line must be a colon, introducing a new descriptor.

The descriptors listed in the table “Capabilities Recognized By LynxOS” are those used by LynxOS utility programs. Others may freely be added for special characteristics needed by custom applications software. Typically, text editing applications require even more than the default ones.

Within string descriptors, the special sequences from the table "Capabilities Recognized By LynxOS" are recognized by the `tgetstr` routine.

Table 2: Special Sequences Within String Descriptors

Special Sequence	Meaning
`\E`	The ASCII escape character, control-`[`.
`\n`	Newline (linefeed).
`\r`	Carriage return.
`\f`	Form feed.
`\t`	Tab.
`\b`	Backspace.
`^X`	Control-X, for any printable character X.
`\nnn`	Character corresponding to the octal number`nnn` .
`\^`	The uparrow (caret) character.
`\\`	The backslash character.

If a colon must be included in a string, it must be written as `\072`. Null characters should be entered as `\200`; the eighth bit is stripped when the strings are printed with `tputs`.

Certain terminals require time delays after some screen control functions, especially those that affect many screen positions. Delays can be specified in string descriptors by following the equals sign with a number, indicating the number of milliseconds of delay needed after the string is sent. The delay value may optionally include a digit following a decimal point. If the delay should be adjusted based upon the number of lines affected, it may be followed by an asterisk. For example, a terminal which requires 7ms of delay after a clear screen operation would have the `cl` string encoded as

```
:cl=7^Z:
```

assuming the clear screen function code is CONTROL-Z. Be warned that many terminal manufacturers supply worst-case delay information which may or may not be reliable. Additionally, individual terminals of a given brand may perform differently than others. Therefore delays should be added pessimistically, or else different entries should be made for “slow” and “fast” instances of the same terminal type.

Delays are implemented by transmitting a number of characters after the control sequence, enough to consume the requested delay time based upon the current baud rate. The character transmitted is that within the global variable `PC`, which should be set by a program using the `termcap` routines. Such programs should also set the `ospeed` global variable to the current output speed code, which can be obtained from the `TIOCGETP` `ioctl` call.

Cursor Motion

The most important cursor motion descriptor is the `cm` string. This string describes the sequence which must be sent to the terminal to position the cursor at any absolute screen location. Because of the diversity of methods used by terminals for this purpose, the `cm` string allows a rather involved encoding of the capability. Within the string, control sequences of the form `%stuff` dictate the placement and treatment of the line and column parameters to the `tgoto` call. The control sequences recognized are:

`%d`	Output line or column as formatted `\ascii` string.
`%2`	Like `%d`, but forces two digits
`%3`	Like `%d`, but forces three digits.
`%.`	Output line or column as one-byte value, like printf `%c` .
`%+x`	Like %. , but value of character x is added prior to output.
`%>xy`	If line or column is greater than `x` , `y` is added; no output is generated.
`%r`	Reverses processing order of line and column; no output is generated.
`%i`	Increments line and column by one; no output is generated.
`%n`	Exclusive-or line and column with octal 140; no output is generated.

`%B`	Line and column are formatted in BCD; not output is generated (used in conjunction with `%.`).
`%D`	Line and column are coded according to the formula `(x - 2 * (x %16))` No output is generated.
`%%`	Outputs a single `%`.

The line and column numbers passed to `tgoto` are processed in that order with the `cm` string. (Note that, strangely, the line and column are passed to `tgoto` in reverse). The `%r` control should be placed at the beginning of the string to reverse the order if necessary.

As an example of a cursor motion string, consider a terminal which requires an escape character followed by an equals sign and two bytes containing the line and column:

```
:cm=\E=%.%.:
```

If the line and column number must be offset by a space, as is commonly the case, the string would be:

```
:cm=\E=%+ %+ :
```

Terminals which use ANSI-“standard” escape sequences would use the string

```
:cm=\E[%i%d;%dH:
```

Which adds one to the line and column numbers (`%i`) because the addresses are based at one instead of zero, then formats the values into printable digits.

The capability strings `up`, `do`, `bc`, and `nd` give sequences to move the cursor up, down, left, and right. The `bc` value is assumed to be control-H if not included in a terminal description. The `am` flag should be included if the terminal moves the cursor to the left edge of the next line after a character is printed at the right margin (autowrap).

Screen Control

The numeric parameters `li` and `co` should be set to the number of lines and columns on the screen. If the “screen” is actually a window on a graphics device, be sure that there is an entry for the particular window size.

The `cd` capability gives the string which clears the screen from the current position to the end of the screen. The `cl` string clears the entire screen and leaves the cursor at the home (left margin, top line) position. The `ce` string clears from the current position to the end of the line. Many terminals support two varieties of area clear functions, some leaving spaces and some leaving nulls. It is usually preferable to choose the one which leaves the cleared area completely empty.

If the terminal can insert a blank line at the current line position, scrolling the last line off the screen, the sequence should be given as the `al` capability. Likewise, the `dl` capability gives the sequence for removing the current line and adding a blank line at the bottom. The `sf` and `sr` strings scroll the screen forwards (up) or backwards (down).

Some terminals support a scrolling region facility rather than simple line insert/delete. Such a facility can be encoded in the `cs` string, whose format is identical to the `cm` string.

There are two common methods of character insertion. The simplest is that once the `ic` string is sent, it inserts a single space into the current line. The other method requires two strings, `im` and `ei`. The `im` string should put the terminal in a mode wherein each subsequent character is inserted into the current line; `ei` takes the terminal out of this mode. If any padding is necessary while in insert mode, the `ip` string should be set to the padding value (normally no real string is associated with `ip`). Similarly, `dm` and `ed` enter and leave a delete mode, which should cause a character to be deleted for each character received in the mode, and `dc` to delete a single character.

Highlighting Text

The `so` and `se` strings enter and leave standout mode, which can be any inverse video, double intensity, blinking, or half intensity mode (blinking and half intensity usually cause more irritation than anything else). Some terminals (various models from Televideo, for example) add a space or two upon entering and leaving standout mode. For such devices, the `sg` numeric parameter should be set to the number of spaces inserted.

An underlining sequence can be given with `us` and `ue`. The `ug` numeric parameter gives the number of extra spaces added when underlining mode is entered or exited.

The audible bell character can be given as the `bl` string. If a non-audible bell is desirable, the `vb` (visible bell) string can be given; usually this causes a screen flash or something else which is obvious to the user.

Initialization

The `is` string should contain any setup sequences which should be sent to the terminal to initialize it. If there is a large amount of data, it can be stored in a file whose name is given with the `if` string. If both `is` and `if` are defined, the `is` string is normally sent first.

"Unique" Terminals

Some terminals support various endearing features which make them like no other. Features such as destructive backspace, inability to scroll, and other strange behavior are by convention encoded as boolean "capabilities" whose names begin with "x".

Sample Description

Here is a description for the fictional Whizzo-100 terminal.

```
whizzo100|Whizzo-100 terminal:\
   :co#80:li#24:\
   :am:\
   :cm=\E=%+ %+ :up=^K:do=^J:nd=^L:\
   :cd=5\ED:cl=5^Z:ce=5\EI:\
   :ic=\EQ :dc=\ER:\
   :so=\EG :se=\EH:
```

The terminal has 24 lines and 80 columns and various control features. It wraps around to column 0 after a character is printed in column 79. The cursor positioning sequence is the common `ESC=row col`, with the row and column offset by 32 (space character). The terminal has clear to end-of-display, clear entire display, and clear to end of line; 5 milliseconds of padding is specified for these. A character can be inserted into the current line with `ESC Q`, or deleted with `ESC R`. Standout mode (probably inverse video) can be entered with `ESC G` and exited with `ESC H`.

If a variant of a particular description is to be created, a new description can be made by giving a new name, a list of capabilities which differ from the original terminal, then a `tc` string which names the original terminal. For example, if a description exists for a Whizzo 100, and an entry is to be created for a 132-column version, it would look like:

```
whizzo100-132:co#132:tc=whizzo100
```

The `tc` entry should be the last one. Any entries before the `tc` entry override those in the original description. If boolean entries which are active in the original must be inactive in the variant, the entry name should be given followed by an “`@`” character:

```
whizzo100-nowrap:am@:tc=whizzo100
```

The trickiest part of creating a new description is the setting of padding values. Many terminals exhibit inconsistent behavior in this area, so a long process of trial-and-error is required before correct values are determined.

Realize that the use of the `termcap` data is entirely up to applications software. LynxOS utilities in general obey the conventions implied above, but there is no way of ensuring that the `ce` string, for example, will only be used to clear a line. Options other than those above may be freely added for the benefit of applications which perform unusual screen functions.

SEE ALSO

Utility Programs - `termcap, tset`

Library Functions - `termcap()`

NAME

ttys – login terminal information

SYNOPSIS

/etc/ttys

#include <ttytab.h>

DESCRIPTION

The file **/etc/ttys** contains lines describing how terminals should be treated by the init and login programs. Each line is of the following form:
special:flag:config:term:login

special names the terminal device, flag is a numeric value which is non-zero if the line is enabled (see below); config is the default configuration type (used by login and tset); term is the default terminal type (for programs using termcap routines); and, finally, login is the pathname of the program to be started by init if the terminal is enabled.

init scans the **/etc/ttys** file when it starts up and also whenever it receives a HUP signal. For each line with a non-zero flag field, init starts a new process with file descriptors 0, 1, and 2 open for stdin, stdout, and stderr. Usually, login is in fact /bin/login, although any executable file will do. The login process is restarted whenever a child process started by init dies.

The getttyent routines return information from the **/etc/ttys** file in the form of ttytab structures, defined in **ttytab.h**.

```
struct ttytab {
  char *tty_name;    /* name of file in /dev */
  int  tty_enable;   /* is tty enabled */
  char *tty_config;  /* default configuration */
  char *tty_type;    /* initial TERM value */
  char *tty_start;   /* pathname of initial (login) appl. */
};
```

The tty table is useful for programs which must determine a configuration or termcap type for a terminal device.

SEE ALSO

Utility Programs - `init, login, tset`

Library Functions - `getttyent(), ttyconfig(), termcap()`

NAME

`uuencode` – format of an encoded `uuencode` file

DESCRIPTION

Files output by `uuencode(1)` consist of a header line, followed by a number of body lines, and a trailer line. The `uudecode(1)` command will ignore any lines preceding the header or following the trailer. Lines preceding a header must not, of course, look like a header.

The header line is distinguished by having the first 6 characters begin & The word begin is followed by a mode (in octal), and a string which names the remote file. A space separates the three items in the header line.

The body consists of a number of lines, each at most 62 characters long (including the trailing newline). These consist of a character count, followed by encoded characters, followed by a newline. The character count is a single printing character, and represents an integer, the number of bytes the rest of the line represents. Such integers are always in the range from 0 to 63 and can be determined by subtracting the character space (octal 40) from the character.

Groups of 3 bytes are stored in 4 characters, 6 bits per character. All are offset by a space to make the characters printing. The last line may be shorter than the normal 45 bytes. If the size is not a multiple of 3, this fact can be determined by the value of the count on the last line. Extra garbage will be included to make the character count a multiple of 4. The body is terminated by a line with a count of zero. This line consists of one ASCII space.

The trailer line consists of end on a line by itself.

SEE ALSO

Utility Programs - `uuencode, uudecode`

HISTORY

The `uuencode` file format appeared in BSD 4.0.

COPYRIGHT

Driver Service Calls

NAME

`Intro –` device driver service calls

DESCRIPTION

The routines described in this section are available to device drivers for performing various services. Normal operating system calls used by applications are, in general, not available to device drivers.

NAME

`alloc_cmem`, `free_cmem` – allocate physically-contiguous memory

SYNOPSIS

```
char *alloc_cmem(size)
int size;

free_cmem(p, size)
char *p;
int size;
```

DESCRIPTION

`alloc_cmem` returns a pointer to a block of physically contiguous memory of at least **size** bytes. If a large enough block is not available, a NULL pointer will be returned. The pointer is a virtual address.

`free_cmem` makes space pointed to by **p** (formerly allocated by `alloc_cmem`) available for further allocation. `free_cmem` does not affect the contents of the space.

NOTE

These calls cannot be used in an interrupt service routine.

SEE ALSO

Driver Calls - `sysbrk(), sysfree()`

NAME

`brelse` – give back disk block

SYNOPSIS

```
#include <disk.h>

brelse(be)
struct buf_entry *be
```

DESCRIPTION

`brelse` will give back the block specified by **be** to the file system. The block will be added to the free block list.

NOTE

This call may not be used in an interrupt routine.

NAME

`cprintf` – formatted output to console

SYNOPSIS

```
cprintf(format, ...)
char *format;
```

DESCRIPTION

`cprintf` is a printing mechanism provided for debugging or reporting serious problems detected by a driver. It works just like `printf`, and sends all output to the current console.

At this time `cprintf` will not print floating point values.

NOTE

This call must not be used in an interrupt routine.

SEE ALSO

Driver Calls - `kkprintf()`

Library Functions - `printf()`

NAME

`deliversigs` – deliver pending signals to a process

SYNOPSIS

`deliversigs()`

DESCRIPTION

`deliversigs` will deliver any pending signals to the process. If there is a user defined signal handler, it is called. The default signal handler is called if there is no user-defined signal handler. `deliversigs` will not return if the user defined signal handler does not return. A signal handler will not return if it exits, longjumps out of the signal handler, or aborts.

NAME

`disable`, `restore` – disable and re-enable interrupts

SYNOPSIS

```
#include<kernel.h>

int ps;
disable(ps);
/* Critical Region */
restore(ps);
```

DESCRIPTION

`disable` and `restore` are used to turn off interrupts and re-enable them for short critical section protection. Once `disable` has been called, no interrupts will occur on the machine until `restore` is called. A side effect of the interrupt masking is that preemptive rescheduling will not occur either. Thus, `disable`/`restore` together allow for fast protection of short critical code sections. When `restore` is called, the `slevel` is checked and rescheduling may occur.

Voluntary rescheduling is allowed while disabled. For instance, if your driver calls `swait` on a locked semaphore while interrupts are disabled, that is acceptable, so long as interrupts are re-enabled via `restore` at the end of the critical section (i.e., the following code is acceptable:

```
disable(ps);
...
swait(&random_semaphore, semaphore_priority);
...
restore(ps);
```

`disable` and `restore` do not inhibit exceptions from occurring on the machine. For instance, accesses to unmapped memory will still result in an exception. Exceptions should be avoided while interrupts are disabled; interrupt and task responsiveness may be unacceptably degraded.

NOTE

Excessive use of `disable` ruins your interrupt and task responsiveness. For long critical sections of code, a semaphore (see `swait`) is more appropriate.

Be extremely careful that, whenever `disable` is called, that `restore` is also called using the same argument as was passed to `disable`. The argument is used to store state information that is required to re-enable interrupts appropriately. Refer to kernel.h for the macro expansion of disable and restore.

`disable`/`restore` can be used within a critical region protected by `sdisable`/`srestore`. However, the converse is not true. If you nest `disable` and `restore` within calls to `sdisable` and `srestore`, make sure that the two facilities use different variables, e.g.,

```
int ps, sps;
...
sdisable(sps);
  /* Preemptive rescheduling disabled */
  ...
disable(ps):
    /* Interrupts disabled */
    ...
  restore(ps);
  ...
srestore(sps);
...
```

On multiprocessors, `disable`/`restore` is implemented as a spin lock for multi-processor safety.

SEE ALSO

Driver Calls - `sdisable(), swait()`

NAME

`dma_lock`, `dma_unlock` – lock and unlock a memory range

SYNOPSIS

```
int dma_lock(tjob, start, size)
int tjob;
int start;
int size;

int dma_unlock(tjob, start, size, dirty)
int tjob;
int start;
int size;
int dirty;
```

DESCRIPTION

`dma_lock` locks down the requested virtual address range into physical memory guaranteeing that the given range will not be paged out by the demand-paged virtual memory subsystem. Memory within the range may then be safely used for operations such as dma. Without using this call, it would be possible for the kernel to page out the given memory area and reassign the underlying physical memory to another virtual address, or to an entire different process after a dma controller had been programmed for I/O into that region. Failure to lock memory using `dma_lock` before doing dma could result in very unpredictable date corruption.

`dma_unlock` undoes the `dma_lock` operation and allows the given region to once again be demand-paged.

start is the starting virtual address of the area to be locked. **size** is the size in bytes of the area to be locked. **tjob** is the process ID of the virtual address space in which start resides. If **dirty** is nonzero, the dirty bit will be set in the pages that are unlocked. **dirty** should be nonzero if the dma operations modified the locked memory so that the demand-paged virtual memory subsystem will know that it must save these pages to the swap area if they are subsequently paged out after the unlock.

NOTES

The granularity of the virtual memory region actually locked is in pages. Therefore, more memory may actually be locked than requested.

The **start** and **size** for an unlock should match the **start** and **size** for the corresponding lock.

These routines should not be called from an interrupt routine.

DIAGNOSTICS

If demand-paged virtual memory is not enabled, both calls simply return `OK`.

`dma_lock` returns `OK` for success, `SYSERR` for failure. The possible failure reasons are:

`ENOMEM`	too many locks or too much locked memory (no more real memory is available; it is all already locked). `dma_unlock` returns OK for success, `SYSERR` for failure. The possible failure reasons are
`EINVAL`	memory was not locked.

SEE ALSO

Utility Programs - `vmstart`

System Calls - `vmstart()`

Driver Service Calls - `mmchain(), vm_lock(), vm_unlock()`

NAME

`get1page, free1page` – get or free one page of physical memory

SYNOPSIS

`char *get1page()`

`free1page(addr)`
`char *addr`

DESCRIPTION

`get1page` allocates one page of physical memory and returns a virtual address which can be used to access it. The virtual address is above `OSBASE`, and is thus valid in the context of any task (in other words, the address can safely be used in an interrupt routine).

Because the memory is aligned on a page boundary, `get1page` is handy for drivers that need a buffer for DMA purposes.

`free1page` releases the page whose virtual address is **`addr`**. It is of course unwise to free a page not allocated with `get1page`.

NOTE

This call may not be used in an interrupt routine.

NAME

`_getpgrp` – get the process group

SYNOPSIS

`_getpgrp()`

DESCRIPTION

`_getpgrp` returns the process group ID of the calling process.

NOTE

This call may *not* be used in an interrupt routine.

NAME

`get_phys` – get physical address of a virtual address

SYNOPSIS

```
char *get_phys(vaddr)
long vaddr;
```

DESCRIPTION

`get_phys` returns the physical address onto which the given virtual address is mapped. The kernel, except for the lowest levels of the memory management subsystem, deals entirely in virtual addresses. However, some hardware devices may require a corresponding physical address if they, for example, access physical memory directly (DMA) without the benefit of the MMU. The value returned by `get_phys` minus `PHYSBASE` may be used for such purposes.

NOTES

The value **vaddr** is not checked for validity. The virtual address must be locked in memory, otherwise the returned address is undefined. To lock memory, use `dma_lock` or `vm_lock`, or make the priority of the process higher than the priority of the virtual memory swapper process.

This call may not be used from an interrupt routine.

SEE ALSO

Utility Programs - `vmstart`

System Calls - `vmstart()`

Driver Service Calls - `mmchain(), dma_lock()`

NAME

`getpid` – get the process ID of the current process

SYNOPSIS

```
int getpid ()
```

DESCRIPTION

`getpid` returns the process id of the current process.

NOTES

This is a process ID, not a thread ID, nor a combined process ID and thread ID.

Use of the function doesn't make sense in an interrupt routine since any process may be the current process when an interrupt occurs.

SEE ALSO

System Calls - `getpid()`

NAME

`_getpriority` – get priority of current process

SYNOPSIS

`_getpriority()`

DESCRIPTION

This returns the priority value of current process. A priority is a number from 0 (lowest) to 255 (highest). The process with the highest priority that is not waiting on a system resource or I/O is by definition the current process.

NOTE

This call may be used in an interrupt routine.

SEE ALSO

System Calls - `getpriority()`

NAME

`hgetime` – get the time of day

SYNOPSIS

`unsigned long hgetime()`

DESCRIPTION

`hgetime` returns the time of day in seconds by reading directly from the hardware clock. A faster way to get the time in seconds is to use `time`. `time` simply reads a global variable which is incremented each second by the clock interrupt routine.

SEE ALSO

Library Function - `time()`

NAME

`htimeout`,
`cancelhtimeout` – high-speed interval timing

SYNOPSIS

```
int htimeout(func, arg, interval)
int(*func)();
char *arg;
int interval;

cancelhtimeout(num)
int num;
```

DESCRIPTION

A call to `htimeout` causes the function **`func`** to be called with one argument, **`arg`**, every **`interval`** milliseconds. `htimeout` returns a positive `timeout` number if there is a timer available.

An operating interval timer can be canceled with `cancelhtimeout` by passing it the `timeout` number returned by `htimeout`.

NOTE

Unlike `timeout`, where the routine is called once after the interval delay, `htimeout` causes the routine to be called repeatedly until it is canceled.

`htimeout` relies on the presence of an auxiliary hardware timer and is only implemented for Motorola MVME-167(3 timers) and MVME-162(5 timers) boards.

These calls may not be used in an interrupt routine.

SEE ALSO

Driver Calls - `timeout()`

NAME

`ioint_link` – share an interrupt vector with another handler

SYNOPSIS

```
ioint_link(id)
int id;
```

DESCRIPTION

`ioint_link` is used on hardware where two or more drivers must share the same interrupt vector. To share an interrupt, each interrupt routine must check if its device has interrupted and then act accordingly; it must also call the next interrupt routine on the list for this hardware vector.

EXAMPLE

In the install routine:

```
next_interrupt_routine = iointset(...);
```

In the interrupt routine:

```
if (my hardware has interrupt pending)
  handle interrupt;
ioint_link(next_interrupt_routine);
```

NAME

`iointset, iointclr` – hardware interrupt dispatch

SYNOPSIS

```
iointset(vector, function, argument)
int vector;
int(*function)();
char *argument;

iointclr(vector)
int vector;
```

DESCRIPTION

Device drivers that must handle interrupts must use `iointset` to inform the interrupt dispatcher what routine to run when an interrupt is received. The interrupt vector number used by the interrupting device is given by **`vector`**. The interrupt dispatcher will call **`function`** when an interrupt is received. The dispatcher passes **`argument`** to the handler, usually a pointer to some sort of device information block created upon device installation.

`iointset` should be called in the device install routine of the device driver. Interrupt dispatch entries should be cleared in the device uninstall routine using `iointclr`, which takes the **`vector`** as its only argument.

For each vector, the kernel maintains a stack of interrupt handlers. Thus, if a driver installs a new handler at a position occupied by an existing handler, the old handler will be re-installed when `iointclr` is called for that vector. (This is not to say that all handlers are invoked upon receipt of an interrupt; the handler functions and arguments are simply saved and restored. Only the last handler installed is called when the interrupt occurs.)

The default “handler” for interrupts is code that causes the kernel to panic and halt.

NOTE

On Intel x86-based systems, these routines require the interrupt vector number, not the address of the vector. Additionally, note that on x86 systems the hardware `IRQ` values should be offset by 32. Thus, a handler for `IRQ5` would be installed as follows:

```
iointset(32 + 5, function, argument);
```

On Motorola 68k-based systems, the interrupt vector value should be multiplied by four when calling iointset and iointclr. In other words, if a device is programmed to interrupt at vector 103, `iointset` would be called as follows:

```
iointset(103 * 4, function, argument);
```

These calls may not be used from an interrupt routine.

On Intel x86-based systems and on SPARC systems `iointset` returns the index of the old interrupt vector. This can be used with `ioint_link` to share interrupt vectors.

NAME

`_kill, _killpg` – signal a process from a driver

SYNOPSIS

```
_kill(pid, signal)
int pid;
int signal;

_killpg(pgrp, signal)
int pgrp;
int signal;
```

DESCRIPTION

`_kill` sends **`signal`** to the process with ID **`pid`**. `_killpg` send signal to process group with ID **`pgrp`**. `_kill` (`_killpg`) must be used instead of `kill` (`killpg`) to send signals from drivers. Fascinating operating system crashes result if the normal system calls are mistakenly used.

NOTE

These calls may be used from an interrupt routine.

There is no way to deliver a signal to a particular thread within a process using `_kill`. Some system calls support the combining of a process ID and a thread ID to allow a signal to be directed at a particular thread. `_kill` *does not support this functionality.* A combined process/thread ID will be seen by `_kill` as a bogus process ID.

In order to guarantee that a particular thread within the process receives a signal, block the signal using `sigprocmask`, and have the desired thread then await the signal's arrival using `sigwait`. This is, as well, the recommended method of using signals and threads in general.

DIAGNOSTICS

Returns `OK` on success. Returns `SYSERR` on failure. Possible failure reasons are:

`EINVAL`	Process ID or signal ID out of range.

`ESRCH` non-existent process ID

SEE ALSO

System Calls - `kill(), killpg(), sigaction(), sigwait()`

Driver Calls - `stkill()`

NAME

`kkprintf` – kernel `printf`

SYNOPSIS

```
kkprintf (format, ...)
char *format;
```

DESCRIPTION

`kkprintf` is a printing mechanism provided for debugging. It works just like `printf`, and sends all output to a fixed device, usually a UART. `kkprintf` uses polling to communicate with the UART so messages may be printed while in an interrupt routine. It will also not return until the entire message is printed.

At this time, `kkprintf` will not print floating point numbers.

NOTE

This call may be used in an interrupt routine, although such use is dangerous.

Note that if the operating system configuration includes a device which uses the hardware used by `kkprintf`, programs which access the device may hang the system.

SEE ALSO

Library Functions - `printf()`

NAME

mem_lock – guarantees address range is in real memory

SYNOPSIS

```
mem_lock(tjob,start,size)
int tjob;
int start;
unsigned long size;
```

DESCRIPTION

tjob is the process ID that wants to lock memory.

start is the virtual start address, in bytes

size is the size of the region of memory to lock, in bytes.

`mem_lock` guarantees that a region of virtual memory is locked into physical memory. `mem_lock` returns `OK` if it can lock all the requested memory otherwise it will return `SYSERR`.

If the kernel is not running virtual memory, `mem_lock` will always return `OK`.

The following is an example of using `mem_lock` in one of our device drivers:

```
if (mem_lock (tpid=getpid(), buff,(long)i<<9)) { ...
```

NOTE

This call may not be used from an interrupt routine.

SEE ALSO

Driver Calls - `mem_unlock()`

NAME

`mem_unlock` – frees real memory for swapping

SYNOPSIS

```
mem_unlock(tjob,start,size,dirty)
int tjob;
char *start;
long size;
int dirty;
```

DESCRIPTION

`tjob` is the process ID that wants to lock memory.

`start` is the virtual start address, in bytes

`size` is the size of the region of memory to lock, in bytes.

if **`dirty`** is non `0` then the entire section of memory from start to size will be marked as dirty. Dirty pages will be written to disk by the `VM` mechanism in the kernel before they are released.

`mem_unlock` unlocks the previously locked region of virtual memory which allows the memory to be swapped. `mem_unlock` returns OK if the region has been successfully unlocked otherwise it will return `SYSERR`. The following is an example of using `mem_lock` in one of our device drivers:

```
mem_unlock(getpid(), buff,(long)bytes,read));
```

NOTE

This call may not be used from an interrupt routine.

If the kernel is not compiled with VM defined, this routine will always return `OK`.

SEE ALSO

Driver Calls - `mem_lock()`

NAME

mmchain, mmchainjob – get physical addresses for DMA chaining

SYNOPSIS

```
mmchain(array, vaddr, size)
struct dmachain *array;
char *vaddr;
long size;

mmchainjob(job, array, vaddr, size)
int job;
struct dmachain *array;
char *vaddr;
long size;
```

DESCRIPTION

mmchain writes into **array** the physical addresses and sizes of all memory pages which make up a contiguous virtual memory segment. The virtual segment is described by its starting address, **vaddr**, and **size** in bytes, **size**. When using mmchain, the virtual address refers to the memory space owned by the current process. When using mmchainjob, the virtual address refers to the memory space owned by the process whose process ID is **job**. **array** should be large enough to hold (**size**+PAGESIZE-1)/PAGESIZE+1 **dmachain** structures.

Each page is described by a **dmachain** structure (defined in mem.h), which has the following form:

```
struct dmachain {
   long address;      /* page physical address */
    short count;      /* page size */
};
```

Both mmchain and mmchainjob return the number of elements written into **array**.

mmchain() and mmchainjob() adds PHYSBASE to address so that the physical address for a DMA controller is *address*–PHYSBASE.

There is no guarantee that the physical memory underlying a contiguous range of virtual addresses is, itself, contiguous.

Virtual pages accessed by their physical addresses must be locked in memory. This can be done by running the process at a priority higher than the virtual memory swapper process, or by locking the memory down using `mem_lock`.

Job is a process ID, not a thread ID or a combined thread/process ID. No checking is done on job; passing in an invalid job is an easy way to crash the system.

NOTE

These calls may not be used from an interrupt routine.

SEE ALSO

Driver Calls - `mem_lock()`, `mem_unlock()`

NAME

NOT_ALIGNED – check for potential alignment faults

SYNOPSIS

#include <kernel.h>

int NOT_ALIGNED(addr)
long *addr;

DESCRIPTION

The slightly misnamed macro, NOT_ALIGNED returns non-zero if accessing the given **addr** as a long would cause a memory alignment fault (not ok to dereference and use). If a memory alignment fault would not occur, a zero value is returned (ok to de-reference and use). It does not return whether or not **addr** is aligned on a long boundary.

```
long l;

...
  if (NOT_ALIGNED(addr)) {  /* Check if deref ok*/
    pseterr(EFAULT);  /* No, would fault*/
    return SYSERR;
  }
  l = *addr;          /* Ok to get value as long*/
...
```

NOTES

In the past, NOT_ALIGNED has been used to check a parameter passed to the kernel by a user-level process before it is de-referenced as a long. This way, the kernel can return EFAULT or bcopy that data as outlined below instead of causing a memory fault. Almost always, it was used when dealing with the third parameter of an ioctl system call to a driver where the kernel itself cannot verify the validity of the data passed in this parameter.

Since it is driver-specific, it is up to the driver to assure correctness.

Use of this macro is discouraged. Since it only checks one case (that of the accessibility of a long), it is incomplete. The appropriate way to access potentially incorrectly aligned user data (including structures, shorts, integers, etc.), after doing the appropriate `rbounds` or `wbounds` check, is to `bcopy` the data pointed to by **`addr`** to a kernel-internal data structure of the appropriate type and then use that variable instead. Thus, the kernel is assured that the data is correctly aligned and the user is free to pass in misaligned data without getting an `EFAULT`. Don't forget to `bcopy` value(s) which should be updated back to the user's area.

In an older version of the macro, not only was it misnamed, it was also misspelled as `NOT_ALLIGNED`.

SEE ALSO

System Calls - `ioctl()`

Library Functions - `bcopy()`

Driver Service Calls - `rbounds()`, `wbounds()`

NAME

`permap` – peripheral mapping for I/O

SYNOPSIS

```
char *permap(physaddr, size)
long physaddr;
long size;
```

DESCRIPTION

The `permap` routine is used to map an area of peripheral memory in the address space of the kernel for access by device drivers. The maximum amount of memory which can be mapped using this call is 64MB.

The **`physaddr`** specifies the physical address of the peripheral device which is to be mapped. **`size`** refers to the number of bytes to map. It should be a multiple of `PAGESIZE`. `permap` returns the virtual address where the physical address has been mapped in. This allows the virtual address to be used to access the peripheral no matter what the current job is.

Use `permap` in the `install` routine of the device driver to map in the required memory. `permap` maps the physical address in a 64MB region just below `PHYSBASE`.

NOTES

There is no way of unmapping the memory mapped using `permap`. This system call is supported only on x86 platforms.

NAME

`pgeterr, pseterr,`
`geterr, seterr` – get and set kernel-level errno

SYNOPSIS

```
int pgeterr()
int pseterr(err)
int err;

#include <kernel.h>

int geterr()

int seterr(err)
int err;
```

DESCRIPTION

pgeterr and **pseterr** are used to get and set the kernel error return code (**uerror**). The value of **uerror** is directly reflected in the **errno** global variable set in user-level programs as the result of system calls.

In general, system call success produces a return value of **0**, while failure gives a **-1** and sets the global variable **errno** to inform the user of the exact reason for the system call failure. The set of possible values for **errno** and their mnemonics can be found in **errno.h**.

When a device driver routine has an error condition that must be passed up to the user, it should use **pseterr** to set **uerror** and return **SYSERR** to its calling routine. Conversely, when it completes successfully, it should return **OK**. Higher-level kernel routines that call device driver entry point routines expect them to follow this convention.

pgeterr returns the current value of **uerror**. **pseterr** sets **uerror** to the value of **err** and returns that value.

geterr and **seterr** are macros. **pgeterr** and **pseterr** are functions. In order to use `geterr` and `seterr`, `kernel.h` must be included.

NOTES

No errno variable in LynxOS

errno exists only in user space. There is no variable called **errno** in the LynxOS kernel.

For example,

```
...
/* Does the pointer the user passed to the kernel point to
enough valid memory in his virtual address space to hold
an int-sized value? */
if (rbounds(p->arg) < sizeof(int)) {
  seterr(EFAULT);      /* No, indicate a memory fault*/
  return SYSERR;     /* return indicating error*/
}
...
return OK;
```

Or,

```
...
if (currpptr->uid) {       /* Only superuser may pass...*/
  seterr(EPERM);
  return SYSERR;
  }
...
return OK;
```

Device Drivers

For device driver writers, be aware that a higher level kernel routines is free to modify the value of **uerror** to make it conform to its specifications. However, if your driver code sets **uerror** and returns **SYSERR**, *some* error will be reflected to the user.

Other Notes

- The name **uerror** is used for simplicity in this discussion.
- The location of the actual value is thread specific and is opaque.

- Only **pgeterr** and **pseterr** should be used to examine and alter its value.
- These routines should not be used from a interrupt routine.

SEE ALSO

System Calls -	(any with error return values)
Documents -	*Device Driver Development Guide*
	ISO/IEC 9945-1 or IEEE 1003.1 (POSIX) -- for **errno** discussion

NAME

`priot_add`, `priot_remove`, `priot_max`, `priot_addn`, `priot_removen` – tracking priorities

SYNOPSIS

```
#include <time.h>

#include <st.h>

priot_add(p, prio)
struct priotrack *p;
int prio;

priot_remove(p, prio)
struct priotrack *p;
int prio;

priot_max(p)
struct priotrack *p;

priot_addn(p, prio, n)
struct priotrack *p;
int prio;
int n;

priot_removen(p, prio, n)
struct priotrack *p;
int prio;
int n;
```

DESCRIPTION

`priot_add` increments the count (number of processes being tracked for a given priority) and sets the bit in the mask corresponding to **`prio`** priority. This returns the highest priority tracked so far, using which the system thread will raise its priority if required.

`priot_remove` decrements the count (number of processes being tracked for a given priority) and reset the bit in the mask of given **`prio`** if `count` becomes less than or equals to 0.

`priot_max` returns `0` if no bit is set in the mask; else returns the highest priority tracked.

`priot_addn` is similar to `priot_add`, except that the count is incremented by **n** instead of `1`.

`priot_removen` is similar to `priot_remove`, except that the count is decremented by **n** instead of `1`.

All these functions will use and update the fields in structure **p** for priority tracking.

NOTE

System threads planning to do priority tracking should raise or lower its priority based on the priority returned by these functions, rather than the **prio** passed as parameter.

SEE ALSO

Driver Calls -	`ststart(), stremove(), stsetprio()`
Utility Programs -	`stasks`

NAME

`rbounds, wbounds` – check pointer validity

SYNOPSIS

```
long rbounds(addr)
unsigned long addr;

long wbounds(addr)
unsigned long addr;
```

DESCRIPTION

`rbounds` returns the number of bytes to the next boundary of non-readable memory in the virtual address space of the calling process. In other words, it returns the number of bytes readable starting at **`addr`**:

- If the address is in the text segment, the distance from **`addr`** to the end of the text segment is returned.
- If the address is in the BSS, the distance from **`addr`** to the current break is returned.
- If the address is in the user stack, the distance to the beginning (top) of the stack is returned.
- If the address is in a shared memory segment, the distance to the end of that shared memory segment is returned.
- If the address lies anywhere else (between the break and the current stack pointer, for example), `rbounds` returns 0.

`wbounds` returns the number of bytes to the next boundary of non-writable memory in the virtual address space. This works similarly to `rbounds`, except that checking for **`addr`** in the text segment is not done.

These routines must be called by the `ioctl` entry points of drivers to inspect pointers passed by application code. The return value from `rbounds` or `wbounds` should be compared to the size of the object the driver expects to be referenced. The error code `EFAULT` should be returned if **`addr`** is found to be erroneous.

NOTE

These calls may be used in an interrupt routine.

NAME

recoset, noreco – non-local goto on hardware fault

SYNOPSIS

recoset()

noreco()

DESCRIPTION

recoset works like a setjmp except no environment information is saved and the long jump is done by the bus fault handler. recoset returns 0 after setting up the bus fault handler. When a hardware fault is encountered, program execution continues as if returning from recoset with a non-zero return value.

noreco resets the bus fault handler so it no longer jumps to the recoset return address.

EXAMPLE

```
p = <device address>
if (!recoset())
  { x = *p;
}
else {      /* no device */
  ...
}
noreco();    /* don't catch jump anymore */
```

NAME

scount – get the semaphore count

SYNOPSIS

```
int scount(sem)
int *sem;
```

DESCRIPTION

scount returns the value of the semaphore **sem**. If the count is negative, it indicates the number of processes waiting on the semaphore.

SEE ALSO

System Calls - sem_signal(), sem_wait()

Library Functions - csem_signal(), csem_wait()

NAME

`sdisable`, `srestore` – disable and re-enable context switching

SYNOPSIS

```
#include <kernel.h>

int ps;
sdisable(ps); /* Critical Region */
srestore(ps);
```

DESCRIPTION

`sdisable` and `srestore` are used to inhibit preemptive rescheduling short critical section protection. Once `sdisable` has been called, the processor will not be preemptively taken away from the calling thread except for interrupt handling. Interrupts still occur, but no other thread will be allowed to run as a result of those interrupts (or as a result of anything the calling thread does, either). `srestore` re-enables preemptive context switching. When `srestore` is called, the `slevel` is checked and rescheduling may occur. Thus, `sdisable`/`srestore` together allow for fast protection of short critical code sections.

Voluntary rescheduling is allowed while sdisabled. For instance, if your driver calls `swait` on a locked semaphore while context switching is inhibited, that is acceptable, so long as context switching is re-enabled via `srestore` at the end of the critical section (i.e., the following code is acceptable):

```
sdisable(ps);
...
swait(&random_semaphore, semaphore_priority);
...
srestore(ps);
```

`sdisable` and `srestore` do not inhibit exceptions from occurring on the machine. For instance, accesses to unmapped memory will still result in an exception. Exceptions should be avoided while context switching is disabled; task responsiveness may be unacceptably degraded.

NOTE

Excessive use of `sdisable` ruins your task responsiveness. For long critical sections of code, a semaphore (see `swait(9)`) is more appropriate.

Be extremely careful that, whenever `sdisable` is called, that `srestore` is also called using the same argument as was passed to `sdisable`. The argument is used to store state information that is required to re-enable context switching appropriately.

`disable`/`restore` can be used within a critical region protected by `sdisable`/`srestore` (make sure that `disable`/`restore` and `sdisable`/`srestore` use different parameters). However, the converse is not true.

On multiprocessors, `sdisable`/`srestore` is implemented as a spin lock for multi-processor safety.

SEE ALSO

Driver Calls - `disable(), swait()`

NAME

`setjob` – set the current job number

SYNOPSIS

```
setjob(x)
int x;
```

DESCRIPTION

Map a task's pages by setting the current job to the task's job number. The old task's pages are no longer mapped. Note that the kernel address space for all tasks is the same; so, the kernel space accesses are not affected by `setjob`.

NAME

`ssignal`, `ssignaln`,
`sreset` – signal a system semaphore

SYNOPSIS

```
ssignal(sem)
int *sem;

ssignaln(sem, count)
int *sem;
int count;

sreset(sem)
int *sem;
```

DESCRIPTION

`ssignal` increments a system semaphore and wakes up one process that is waiting on that semaphore. Processes are awakened in priority order.

`ssignaln` signals the semaphore **`count`** times.

`sreset` wakes up all processes which are waiting on a semaphore and sets the semaphore value to zero.

NOTE

Signaling a semaphore has nothing to do with sending a process a signal. These calls may be used from an interrupt routine.

SEE ALSO

Driver Calls - `swait()`

NAME

`st_getstid` – get the system thread ID of the current thread

SYNOPSIS

`int st_getstid()`

DESCRIPTION

`st_getstid` returns the process ID of the current system thread. A process, when first created, has one system thread in it. POSIX threads are implemented using system threads, as are the stream tasks used in Lynx drivers. Calling `st_getstid` at any time, therefore, yields a valid result, except when handling an interrupt (interrupts are handled in the context of whichever thread happened to be running when the interrupt occurred).

NOTES

This is a system thread ID, not a process ID, nor a combined process and thread ID.

Use of the function does not make sense in an interrupt routine since any thread may be the current thread when an interrupt occurs.

SEE ALSO

Driver Service Calls - `getpid()`

NAME

`strlog` – STREAMS error logging and tracing

SYNOPSIS

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

strlog(m_id, s_id, level, flags, format, ...)
short m_id;
short s_id;
char level;
char *format;
```

DESCRIPTION

`strlog` is a STREAMS driver that gives an interface for LynxOS kernel error messages, and for STREAMS errors. `strerr` and `strace` use `strlog` to get their event traces.

From the kernel side, the routine `strlog` is used to log traces to the `strlog` driver. From the user application side, a set of ioctls are used to operate on the messages which are currently in the log.

The kernel interface, `strlog`(), is called with the following parameters:

m_id	The module ID of the module logging the message
s_id	The sub-ID of the minor device within the module.
level	a priority level for the message, which can be used to screen out less-important messages.
flags	various flags affecting logging:
SL_ERROR	log to the error message log
SL_TRACE	log to the trace log
SL_FATAL	fatal error
format	a format string in the style of printf, used to format the message for the log. Arguments to the format string follow the format string, just like a call to printf.

From the user side, the device `/dev/strlog` is opened to interrogate and operate on `strlog`.

NAME

`ststart`, `stremove`,
`stsetprio`, `stgetprio`,
`stgetid`, `stgetowner`,
`stkill`, `stsignal` – handlers for system threads

SYNOPSIS

```
int ststart(procaddr, stacksize, prio, name, nargs, arg1,
arg2, ...)
int(*procaddr)();
int stacksize;
int prio;
char *name;
int nargs;
char *args;

void stremove(stid)
int stid;

void stsetprio(stid, prio)
int stid;
int prio;

void stgetprio(stid)
int stid;

int stgetid()

void stgetowner(stid)
int stid;

void stkill(stid, signal)
int stid;
int signal;

void stsignal(handler, arg)
int (*handler)();
char *arg;
```

DESCRIPTION

System threads are used by the kernel and device drivers to perform an asynchronous activity with a given priority. A system thread allows time-consuming activities to be done in a schedulable and preemptable way, instead of in an interrupt routine.

`ststart` starts a system thread using code beginning at **`procaddr`** with given priority **`prio`**. The priority should be in the range `0...512`. If the priority is even, the thread will be scheduled at system priority level **`prio`**÷2. If the priority is odd, then the thread will be scheduled "between" levels **`prio`**÷2 and **`prio`**÷2+1. In other words, a thread with an odd prio will be preferentially scheduled ahead of other tasks at priority **`prio`**÷2.

The thread is given **`stacksize`** bytes of stack, which will not be expanded. Care should be taken to ensure that the allocated amount is sufficient, but not excessive. The **`nargs`** arguments beginning with **`arg1`** are passed to the stream task. For internal recording purposes, a pointer to an identifying string should be passed in **`name`**.

`If successful`, `ststart` returns an integer, the **`stid`** ("stream task ID") which should be saved for later use, otherwise `SYSERR` (`-1`) is returned.

`stremove` removes the system thread from the system and releases its stack space. Memory allocated by the thread with `sysbrk` or `get1page` is not released when the thread is removed. No error is set and no value is returned by `stremove`. **`stid`** must be greater than or equal to `0` and less than `NSTASKS`; a value of **`stid`** outside of this range will crash the system.

`stsetprio` sets the priority of the given system thread **`stid`** to **`prio`**. The priority value is interpreted as described above. No error is set and no value is returned by `stsetprio`.

`stgetprio` returns the priority of the given system thread **`stid`**. The priority value is interpreted as described above. No error is set by `stgetprio`.

`stgetid` returns the stream task id of the calling thread. No error is set by `stgetid`.

`stkill` sends **`signal`** to stream task **`stid`**. **`signal`** must be in the range 1-31.

`stsignal` sets up a signal handler for the stream task. It must be called from the stream task because stid is not passed to `stsignal`. When **`handler`** is called **`arg`** and **`signal`** (from `stkill`) are passed to **`handler`**. There can be only one handler per stream task. **`handler`** is signaled by calling `stkill`. No error is set and no value is returned by `stsignal`.

These calls may not be used in an interrupt routine.

DIAGNOSTICS

ststart returns stid if successful, or -1 if not.

If ststart fails, **`errno`** will contain one of the following:

`EAGAIN`	no more system threads available.
`ENOMEM`	`memory requested not available for thread.`

NAME

`swait` – wait on a system semaphore

SYNOPSIS

```
swait(sem, flag)
int *sem;
int flag;
```

DESCRIPTION

`swait` causes the calling process to wait on a semaphore. A semaphore is simply a pointer to an integer; it is allocated by the driver and needs no special initialization other than to set its initial value. `swait` first decrements the integer. If the integer is negative after it is decremented, the process is removed from the “ready” list and put into a queue. Thus a negative semaphore count represents the number of processes that are waiting on the semaphore, and a positive count represents the number of times the semaphore can be “waited” on without having to wait for the semaphore to be signaled. A semaphore must be initialized to some non-negative value before use.

The **flag** argument determines whether or not signals will be delivered to the process while it is waiting. If **flag** is zero, all pending signals will be delivered to the process while it waits. The process will automatically re-wait on the semaphore after a signal is handled. If **flag** is less than zero, no signals will be delivered while the process waits. If **flag** is greater than zero, a signal will cause the `swait` call to abort and return a non-zero value.

NOTE

The semaphore must be in the supervisor memory space and not on the stack.

This call may not be used from an interrupt routine.

SEE ALSO

Driver Calls - `ssignal()`

NAME

sysbrk, sysfree – dynamic memory allocation

SYNOPSIS

```
char *sysbrk(size)
long size;

sysfree(p, size)
char *p;
long size;
```

DESCRIPTION

sysbrk returns a pointer to the beginning of a free memory segment of **size** bytes. This memory can be returned to the free list by calling sysfree with the pointer, **p**, and the size.

A device driver usually allocates memory in its device install routine for device-specific information and should free that memory in its device uninstall routine. Memory may be allocated and deallocated in other device driver routines also.

NOTE

These calls may not be used from an interrupt routine.

NAME

`timeout, cancel_timeout` – interval timing

SYNOPSIS

```
int timeout(func, arg, interval)
int(*func)();
char *arg;
int interval;

cancel_timeout(num)
int num;
```

DESCRIPTION

A call to `timeout` causes the function **`func`** to be called with one argument, **`arg`**, after **`interval`** hundredths of a second. `timeout` returns a positive timeout number if there is a timer available.

A `timeout` can be canceled before the routine is called using `cancel_timeout` and passing it the `timeout` number.

NOTE

Care should be taken to cancel only a pending `timeout`.

These calls may be used in an interrupt routine, but should not be used in a device install routine.

SEE ALSO

Driver Calls - `htimeout()`

NAME

`tmgr_close, tmgr_mclose` – close tty

SYNOPSIS

```
#include <ttymgr.h>
#include <file.h>

tmgr_close(ttyinfo, fd)
struct ttystatics *ttyinfo;
struct file *fd;

tmgr_mclose(ttyinfo, fd)
struct ttystatics *ttyinfo;
struct file *fd;
```

DESCRIPTION

`tmgr_close` and `tmgr_mclose` closes the `tty` for further operations like, `read` and `write`. `tmgr_mclose` is for pseudo-`tty` interface.

`ttyinfo` is the structure initialized by `tmgr_install` and **`fd`** is the file interface structure initialized by `tmgr_open` (`tmgr_mopen`).

NOTE

Each pseudo-tty on the system has its own major device number and two nodes in the `/dev` directory. The nodes share the same `major number` and are distinguished by minor numbers 0 and 1. The node with `minor number` 1 is called the slave and the node with `minor number` 0 is called the master. `tmgr_mclose` is called for the master node.

SEE ALSO

System Calls - `ioctl()`

Driver Calls - `tmgr_install()`, `tmgr_open()`, `tmgr_read()`, `tmgr_write()`, `tmgr_ioctl()`, `tmgr_select()`, `tmgr_tx()`

Library Functions - `ttyname()`

Utility Programs - `tset`, `tty`, `stty`

NAME

`tmgr_install`,
`tmgr_uninstall` – install/uninstall tty

SYNOPSIS

```
#include <ttymgr.h>
#include <ioctl.h>

tmgr_install(ttyinfo, sg, ispty, transmit_enable, arg)
struct ttystatics *ttyinfo;
struct old_sgttyb *sg;
int ispty;
int(*transmit_enable)();
char *arg;

tmgr_uninstall(s)
struct ttystatics *s;
```

DESCRIPTION

`tmgr_install` installs one tty by initializing the lines and required data-structures. **`ttyinfo`** contains current settings of this tty and will be used by other tty manager calls. **`sg`** contains values for various settings to be used for initializing this tty. **`ispty`** tells whether the current tty is a pseudo-tty or a real one. **`transmit_enable`** is a function that will be called by the tty manager to enable character transmissions to the tty hardware. **`arg`** is the argument passed to **`transmit_enable`** when called.

`tmgr_uninstall` "un-installs" the tty. The information about the tty is taken from **`ttyinfo`**, which is initialized by `tmgr_install`.

NOTE

In a driver, which takes care of multiple ttys (whose `minor number` differs), `tmgr_install` and `tmgr_uninstall` have to be called once for each tty.

SEE ALSO

System Calls -	`ioctl()`
Driver Calls -	`tmgr_open()`, `tmgr_close()`, `tmgr_read()`, `tmgr_write()`, `tmgr_ioctl()`, `tmgr_select()`, `tmgr_tx()`
Library Functions -	`ttyname()`
Utility Programs -	`tset`, `tty`, `stty`

NAME

`tmgr_ioctl`, `tmgr_mioctl` – tty I/O control

SYNOPSIS

```
#include <ttymgr.h>
#include <file.h>

tmgr_ioctl(ttyinfo, fd, com, arg)
struct ttystatics *ttyinfo;
struct file *fd;
int com;
char *arg;

tmgr_mioctl(ttyinfo, fd, com, arg)
struct ttystatics *ttyinfo;
struct file *fd;
int com;
char *arg;
```

DESCRIPTION

`tmgr_ioctl` changes various tty parameters. **com** is the function code. **arg** is a pointer to data and its form depends on the particular function. (See tty in Device Facilities for list of available functions.)

`tmgr_mioctl` is for the pseudo-tty interface.

ttyinfo is a structure initialized by `tmgr_install` and **fd** is the file interface structure initialized by `tmgr_open` (`tmgr_mopen`).

NOTE

Each pseudo-tty on the system has its own major device number and two nodes in the `/dev` directory. The nodes share the same `major number` and are distinguished by minor numbers 0 and 1. The node with `minor number` 1 is called the slave and the node with `minor number` 0 is called the master. `tmgr_mioctl` is called for the master node.

SEE ALSO

System Calls -	`ioctl()`
Driver Calls -	`tmgr_install()`, `tmgr_open()`, `tmgr_close()`, `tmgr_read()`, `tmgr_write()`, `tmgr_select()`, `tmgr_tx()`
Device Facilities -	`tty`
Library Functions -	`ttyname()`
Utility Programs -	`tset`, `tty`, `stty`

NAME

`tmgr_open, tmgr_mopen` – open tty

SYNOPSIS

```
#include <ttymgr.h>
#include <file.h>

tmgr_open(ttyinfo, fd)
struct ttystatics *ttyinfo;
struct file *fd;

tmgr_mopen(ttyinfo, fd)
struct ttystatics *ttyinfo;
struct file *fd;
```

DESCRIPTION

`tmgr_open` and `tmgr_mopen` opens the tty for further operations like, `read` and `write`. `tmgr_mopen` is for pseudo-tty interface.

`ttyinfo` is the structure initialized by `tmgr_install` and **`fd`** is the file interface structure to access this device by the programs.

NOTE

Each pseudo-tty on the system has its own major device number and two nodes in the `/dev` directory. The nodes share the same `major number` and are distinguished by minor numbers 0 and 1. The node with `minor number` 1 is called the slave and the node with `minor number` 0 is called the master. `tmgr_mopen` is called for the master node.

SEE ALSO

System Calls -	`ioctl()`
Driver Calls -	`tmgr_install()`, `tmgr_close()`, `tmgr_read()`, `tmgr_write()`, `tmgr_ioctl()`, `tmgr_select()`, `tmgr_tx()`
Library Functions -	`ttyname()`
Utility Programs -	`tset`, `tty`, `stty`

NAME

tmgr_read, tmgr_mread – read characters from tty

SYNOPSIS

```
#include <ttymgr.h>
#include <file.h>

tmgr_read(ttyinfo, fd, buf, count)
struct ttystatics *ttyinfo;
struct file *fd;
char *buf;
int count;

tmgr_mread(ttyinfo, fd, buf, count)
struct ttystatics *ttyinfo;
struct file *fd;
char *buf;
int count;
```

DESCRIPTION

tmgr_read and tmgr_mread read the **count** number of characters from the tty and put them into **buf**. Both functions return the number of characters read. The characters read from the tty are processed and put into **buf** according to the modes set in the tty (see tmgr_ioctl).

tmgr_mread is for the pseudo-tty interface.

ttyinfo is the structure initialized by tmgr_install and **fd** is the file interface structure initialized by tmgr_open (tmgr_mopen).

NOTE

Each pseudo-tty on the system has its own major device number and two nodes in the /dev directory. The nodes share the same major number and are distinguished by minor numbers 0 and 1. The node with minor number 1 is called the slave and the node with minor number 0 is called the master. tmgr_mread is called for the master node.

SEE ALSO

System Calls -	`ioctl()`
Driver Calls -	`tmgr_install()`, `tmgr_open()`, `tmgr_close()`, `tmgr_write()`, `tmgr_ioctl()`, `tmgr_select()`, `tmgr_tx()`
Library Functions -	`ttyname()`
Utility Programs -	`tset`, `tty`, `stty`

NAME

`tmgr_select`,
`tmgr_mselect - tty` I/O multiplexing

SYNOPSIS

```
#include <ttymgr.h>
#include <file.h>

tmgr_select(ttyinfo, fd, cmd, selsem)
struct ttystatics *ttyinfo;
struct file *fd;
int cmd;
struct sel *selsem;

tmgr_mselect(ttyinfo, fd, cmd, selsem)
struct ttystatics *ttyinfo;
struct file *fd;
int cmd;
struct sel *selsem;
```

DESCRIPTION

`tmgr_select` will select the tty identified by **`ttyinfo`** for non-blocking read or write as specified by cmd (`SREAD` or `SWRITE`). Another command, `SEXCEPT`, available for `select` system call, will return error. Fields `iosem` and `sel_sem` in structure **`selsem`** are initialized with appropriate queue from **`ttyinfo`**.

`tmgr_mselect` is for pseudo-tty interface.

NOTE

Each pseudo-tty on the system has its own major device number and two nodes in the `/dev` directory. The nodes share the same `major number` and are distinguished by minor numbers 0 and 1. The node with `minor number` 1 is called the slave and the node with `minor number` 0 is called the master. `tmgr_mselect` is called for the master node.

SEE ALSO

Utility Programs -	`tset, tty, stty`
System Calls -	`ioctl()`
Library Functions -	`ttyname()`
Device Facilities -	`tty`
Driver Calls -	`tmgr_install(), tmgr_open(), tmgr_close(), tmgr_read(), tmgr_write(), tmgr_ioctl(), tmgr_tx()`

NAME

tmgr_tx, tmgr_rx,
tmgr_ex – tty character transmitter/ receiver for ISR

SYNOPSIS

```
#include <ttymgr.h>

tmgr_tx(ttyinfo, cp)
struct ttystatics *ttyinfo;
char *cp;

tmgr_rx(ttyinfo, c)
struct ttystatics *ttyinfo;
char c;

tmgr_ex(ttyinfo, cmd)
struct ttystatics *ttyinfo;
int cmd;
```

DESCRIPTION

tmgr_tx retrieves the next character to be transmitted. The character is placed in the character buffer pointed to by **cp**.

tmgr_rx enqueues the next input character, **c**.

Both tmgr_tx and tmgr_rx also take care of the protocol implementation and decide which characters to put into the canonical list and which characters to write to the output buffer.

tmgr_ex is the exception handler, where the commands BREAK and HANGUP are handled.

SEE ALSO

System Calls -	`ioctl()`
Driver Calls -	`tmgr_install()`, `tmgr_open()`, `tmgr_close()`, `tmgr_read()`, `tmgr_write()`, `tmgr_ioctl()`, `tmgr_select()`
Device Facilities -	`tty`
Library Functions -	`ttyname()`
Utility Programs -	`tset`, `tty`, `stty`

NAME

tmgr_write, tmgr_mwrite – write characters to tty

SYNOPSIS

```
#include <ttymgr.h>
#include <file.h>

tmgr_write(ttyinfo, fd, buf, count)
struct ttystatics *ttyinfo;
struct file *fd;
char *buf;
int count;

tmgr_mwrite(ttyinfo, fd, buf, count)
struct ttystatics *ttyinfo;
struct file *fd;
char *buf;
int count;
```

DESCRIPTION

tmgr_write and tmgr_mwrite writes the **count** number of characters from the **buf** to tty. Returns the number of characters written. The characters taken from the **buf** are processed and written to the tty according to the modes set to the tty (see tmgr_ioctl). tmgr_mwrite is for pseudo-tty interface.

ttyinfo is the structure initialized by tmgr_install and **fd** is the file interface structure initialized by tmgr_open (tmgr_mopen).

NOTE

Each pseudo-tty on the system has its own major device number and two nodes in the /dev directory. The nodes share the same major number and are distinguished by minor numbers 0 and 1. The node with minor number 1 is called the slave and the node with minor number 0 is called the master. tmgr_mwrite is called for the master node.

SEE ALSO

System Calls -	`ioctl()`
Driver Calls -	`tmgr_install()`, `tmgr_open()`, `tmgr_close()`, `tmgr_read()`, `tmgr_ioctl()`, `tmgr_select()`, `tmgr_tx()`
Library Functions -	`ttyname()`
Utility Programs -	`tset`, `tty`, `stty`

Master Index

S

T

U

V

W