STREAMS-Based Terminal Subsystem

12

Overview of Terminal Subsystem

STREAMS provides a uniform interface for implementing character I/O devices and networking protocols in the kernel. SunOS 5.3 implements the terminal subsystem in STREAMS. The STREAMS-based terminal subsystem (Figure 12-1) provides many benefits:

- Reusable line discipline modules. The same module can be used in many STREAMS where the configuration of these STREAMS may be different.
- Line-discipline substitution. Although Sun provides a standard terminal line- discipline module, another one conforming to the interface can be substituted. For example, a remote login feature may use the terminal subsystem line discipline module to provide a terminal interface to the user.
- Internationalization. The modularity and flexibility of the STREAMS-based terminal subsystem enables an easy implementation of a system that supports multiple-byte characters for internationalization. This modularity also allows easy addition of new features to the terminal subsystem.
- Easy customizing. Users may customize their terminal subsystem environment by adding and removing modules of their choice.
- The pseudo-terminal subsystem. The pseudo-terminal subsystem can be easily supported (this is discussed in more detail in the section “STREAMS-based Pseudo-Terminal Subsystem" later in this chapter).
- Merge with networking. By pushing a line discipline module on a network line, you can make the network look like a terminal line.

12

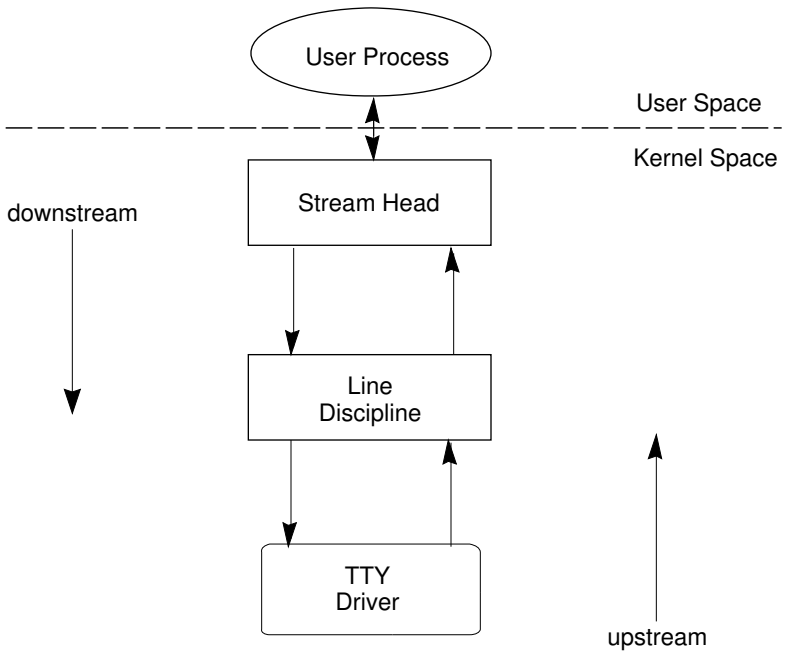

Figure 12-1 STREAMS-based Terminal Subsystem

The initial setup of the STREAMS-based terminal subsystem is handled with the `ttymon`(1M) command within the framework of the Service Access Facility or the autopush feature. See Appendix E, “Configuration" for more information on autopush.

The STREAMS-based terminal subsystem supports `termio`(7), the `termios`(3) specification of the POSIX standard, multiple byte characters for internationalization, the interface to asynchronous hardware flow control (see `termiox`(7)), and peripheral controllers for asynchronous terminals. XENIX® and BSD compatibility can also be provided by pushing the `ttcompat` module.

To use `shl` with the STREAMS-based terminal subsystem, the `sxt` driver is implemented as a STREAMS-based driver. However, the `sxt` feature is being discontinued and users are encouraged to use the job-control mechanism. Note that both `shl` and job control should *not* be run simultaneously.

Line-Discipline Module

A STREAMS line-discipline module called `ldterm` (see `ldterm`(7)) is a key part of the STREAMS-based terminal subsystem. Throughout this chapter, the terms *line discipline* and `ldterm` are used interchangeably and refer to the STREAMS version of the standard line discipline and not the traditional character version. `ldterm` performs the standard terminal I/O processing that was traditionally done through the *linesw* mechanism.

The `termio` and `termios` specifications describe four flags that are used to control the terminal:

- *c_iflag* (defines input modes)
- *c_oflag* (defines output modes)
- *c_cflag* (defines hardware control modes)
- *c_lflag* (defines terminal functions used by `ldterm`).

To process these flags elsewhere (for example, in the firmware or in another process), a mechanism is in place to turn on and off the processing of these flags. When `ldterm` is pushed, it sends an `M_CTL` message downstream that asks the driver which flags the driver will process. The driver sends back that message in response if it needs to change `ldterm`'s default processing. By default, `ldterm` assumes that it must process all flags except *c_cflag*, unless it receives a message telling otherwise.

Default Settings

When `ldterm` is pushed on the Stream, the open routine initializes the settings of the `termio` flags. The default settings are:

c_iflag = `BRKINT|ICRNL|IXON|IMAXBEL`

c_oflag = `OPOST|ONLCR|TAB3`

c_cflag = `CREAD|CS8|B9600`

c_lflag = `ISIG|ICANON|ECHO|ECHOK|IEXTEN|ECHOE|ECHOKE | ECHOCTL`

In canonical mode (`ICANON` flag in *c_lflag* is turned on), `read` from the terminal file descriptor is in message non-discard (`RMSGN`) mode (see `streamio`(7)). This implies that in canonical mode, `read` on the terminal file

descriptor always returns at most one line regardless how many characters have been requested. In non-canonical mode, `read` is in byte-stream (`RNORM`) mode. The flag `ECHOCTL` has been added for SunOS 4.1 compatibility.

User-Configurable Settings

See `termio`(7) for more information.

Open and Close Routines

The open routine of the `ldterm` module allocates space for holding the TTY structure (see `ldtermstd_state_t` in `ldterm.h`) by allocating a buffer from the STREAMS buffer pool. The number of modules that can be pushed on one stream, as well as the number of TTY's in use, is limited. The number of instances of ldterm that have been pushed is limited only by available memory. The open also sends an `M_SETOPTS` message upstream to set the Stream head high and low water marks to 1024 and 200, respectively. These are the current values (although they may change over time).

The `ldterm` module identifies itself as a TTY to the stream head by sending an `M_SETOPTS` message upstream with the `SO_ISTTY` bit of *so_flags* set. The Stream head allocates the controlling TTY on the `open`, if one is not already allocated.

To maintain compatibility with existing applications that use the `O_NDELAY` flag, the open routine sets the `SO_NDELON` flag on in the *so_flags* field of the `stroptions` structure in the `M_SETOPTS` message.

The open routine fails if there are no buffers available (cannot allocate the internal state structure) or when an interrupt occurs while waiting for a buffer to become available.

The close routine frees all the outstanding buffers allocated by this Stream. It also sends an `M_SETOPTS` message to the Stream head to undo the changes made by the open routine. The `ldterm` module also sends `M_START` messages downstream to undo the effect of any previous `M_STOP` messages.

Read-Side Processing

The ldterm module's read side processing has put and service procedures. High and low water marks for the read queue are 1024 and 200, respectively. These are the current values and may be subject to change.

ldterm can send the following messages upstream:

M_DATA, M_BREAK, M_PCSIG, M_SIG, M_FLUSH, M_ERROR, M_IOCACK, M_IOCNAK, M_HANGUP, M_CTL, M_SETOPTS, M_COPYOUT, and M_COPYIN.

The ldterm module's read side processes M_BREAK, M_DATA, M_CTL, M_FLUSH, M_HANGUP, M_IOCACK and M_IOCNAK messages. All other messages are sent upstream unchanged.

The put procedure scans the message for flow-control characters (IXON), signal-generating characters, and after (possible) transformation of the message, queues the message for the service procedure. Echoing is handled completely by the service procedure.

In canonical mode if the ICANON flag is on in *c_lflag*, canonical processing is performed. If the ICANON flag is off, non-canonical processing is performed (see termio(7) for more details). Handling of VMIN/VTIME in the STREAMS environment is somewhat complicated, because read needs to activate a timer in the ldterm module in some cases; hence, read notification becomes necessary. When a user issues an ioctl to put ldterm in non-canonical mode, the ldterm module sends an M_SETOPTS message to the Stream head to register read notification. Further reads on the terminal file descriptor will cause the Stream head to issue an M_READ message downstream and data will be sent upstream in response to the M_READ message. With read notification, buffering of raw data is performed by ldterm. It is possible to canonize the raw data, when the user has switched from raw to canonical mode. However, the reverse is not possible.

To summarize, in non-canonical mode, the ldterm module buffers all data until VMIN or VTIME criteria are met. For example, if VMIN=3 and VTIME=0, and three bytes have been buffered by ldterm, these characters would be sent to the stream head *regardless* of whether there is a pending M_READ, and no M_READ would need to be sent down. If an M_READ message is received, the number of bytes sent upstream will be the argument of the M_READ message, unless VTIME is satisfied before VMIN (for example. the timer has expired) in which case whatever characters are available will be sent upstream.

The service procedure of ldterm handles STREAMS related flow control. Since the read side high and low water marks are 1024 and 200 respectively, placing 1024 characters or more on the ldterm's read queue will cause the QFULL flag be turned on indicating that the module below should not send more data upstream.

Input flow control is regulated by the line-discipline module by generating M_STARTI and M_STOPI high priority messages. When sent downstream, receiving drivers or modules take appropriate action to regulate the sending of data upstream. Output flow control is activated when ldterm receives flow control characters in its data stream. The ldterm module then sets an internal flag indicating that output processing is to be restarted/stopped and sends an M_START/M_STOP message downstream.

Write-Side Processing

Write side processing of the ldterm module is performed by the write-side put and service procedures.

The ldterm module supports the following ioctls:

TCSETA, TCSETAW, TCSETAF, TCSETS, TCSETSW, TCSETSF, TCGETA, TCGETS, TCXONC, TCFLSH and TCSBRK.

All ioctls not recognized by the ldterm module are passed downstream to the neighboring module or driver.

The following messages can be received on the write side:

M_DATA, M_DELAY, M_BREAK, M_FLUSH, M_STOP, M_START, M_STOPI, M_STARTI, M_READ, M_IOCDATA, M_CTL, and M_IOCTL.

On the write side, the ldterm module processes M_FLUSH, M_DATA, M_IOCTL, and M_READ messages, and all other message are passed downstream unchanged.

An M_CTL message is generated by ldterm as a query to the driver for an intelligent peripheral and to decide on the functional split for termio processing. If all or part of termio processing is done by the intelligent peripheral, ldterm can turn off this processing to avoid computational overhead. This is done by sending an appropriate response to the M_CTL message, as follows: (see also ldterm(7)).

- If all of the `termio` processing is done by the peripheral hardware, the driver sends an `M_CTL` message back to `ldterm` with *ioc_cmd* of the structure `iocblk` set to `MC_NO_CANON`. If `ldterm` is to handle all `termio` processing, the driver sends an `M_CTL` message with *ioc_cmd* set to `MC_DO_CANON`. The default is `MC_DO_CANON`.
- If the peripheral hardware handles only part of the `termio` processing, it informs `ldterm` in the following way:

  The driver for the peripheral device allocates an `M_DATA` message large enough to hold a `termios` structure. The driver then turns on those *c_iflag*, *c_oflag*, and *c_lflag* fields of the `termios` structure that are processed on the peripheral device by ORing the flag values. The `M_DATA` message is then attached to the *b_cont* field of the `M_CTL` message it received. The message is sent back to `ldterm` with *ioc_cmd* in the data buffer of the `M_CTL` message set to `MC_PART_CANON`.

One difference between AT&T STREAMS and SunOS 5.x is that AT&T's line discipline module does not check if write side flow control is in effect before forwarding data downstream. It expects the downstream module or driver to add the messages to its queue until flow control is lifted. This is not true in SunOS 5.x.

EUC Handling in ldterm

The idea of letting post-processing (the *o_flags*) happen off the host processor is not recommended unless the board software is prepared to deal with international (EUC) character sets properly. The reason for this is that post-processing must take the EUC information into account. `ldterm` knows about the screen width of characters (that is, how many columns are taken by characters from each given code set on the current physical display) and it takes this width into account when calculating tab expansions. When using multi-byte characters or multi-column characters `ldterm` automatically handles tab expansion (when TAB3 is set) and does not leave this handling to a lower module or driver.

By default, multi-byte handling by `ldterm` is turned off. When `ldterm` receives an `EUC_WSET ioctl` call, it turns multi-byte processing on, if it is essential to properly handle the indicated code set. Thus, if one is using single byte 8-bit codes and has no special multi-column requirements, the special

multi-column processing is not used at all. This means that multi-byte processing does not reduce the processing speed or efficiency of `ldterm` unless it is actually used.

The following describes how the EUC handling in `ldterm` works:

First, the multi-byte and multi-column character handling is only enabled when the `EUC_WSET ioctl` indicates that one of the following conditions is met:

- Code set consists of more than one byte (including the SS2 and/or SS3) of characters
- Code set requires more than one column to display on the current device, as indicated in the `EUC_WSET` structure.

Assuming that one or more of the a previous conditions, EUC handling is enabled. At this point, a parallel array (see `ldterm_mod` structure) used for other information, is allocated and a pointer to it is stored in *t_eucp_mp*. The parallel array that it holds is pointed to by *t_eucp*. The *t_codeset* field holds the flag that indicates which of the code sets is currently being processed on the read side. When a byte with the high bit arrives, it is checked to see if it is SS2 or SS3. If so, it belongs to code set 2 or 3. Otherwise, it is a byte that comes from code set 1. Once the extended code set flag has been set, the input processor retrieves the subsequent bytes, as they arrive, to build one multi-byte character. The counter field *t_eucleft* tells the input processor how many bytes remain to be read for the current character. The parallel array *t_eucp* holds for each logical character in the canonical buffer its display width. During erase processing, positions in the parallel array are consulted to determine how many backspaces need to send to erase each logical character. (In canonical mode, one backspace of input erases one logical character, no matter how many bytes or columns that character consumes.) This greatly simplifies erase processing for EUC.

The *t_maxeuc* field holds the maximum length, in memory bytes, of the EUC character mapping currently in use. The *eucwioc* field is a sub-structure that holds information about each extended code set.

The *t_eucign* field aids in output post-processing (tab expansion). When characters are output, `ldterm` keeps a column to indicate what the current cursor column is supposed to be. When it sends the first byte of an extended character, it adds the number of columns required for that character to the output column. It then subtracts one from the total width in memory bytes of

that character and stores the result in *t_eucign*. This field tells `ldterm` how many subsequent bytes to ignore for the purposes of column calculation. (`ldterm` calculates the appropriate number of columns when it sees the first byte of the character.)

The field *t_eucwarn* is a counter for occurrences of bad extended characters. It is mostly useful for debugging. After receiving a certain number of illegal EUC characters (perhaps because of some problem on the line or with declared values), a warning is given on the system console.

There are two relevant files for handling multi-byte characters: `euc.h` and `eucioctl.h`. The `eucioctl.h` contains the structure that is passed with `EUC_WSET` and `EUC_WGET` calls. The normal way to use this structure is to get `CSWIDTH` from the *locale* via a mechanism such as `getwidth` or `setlocale` and then copy the values into the structure in `eucioctl.h`, and send the structure via an `I_STR ioctl` call. The `EUC_WSET` call informs the `ldterm` module about the number of bytes in extended characters and how many columns the extended characters from each set consume on the screen. This allows `ldterm` to treat multi-byte characters as single units for the purpose of erase processing and to correctly calculate tab expansions for multi-byte characters.

Note – `LC_CTYPE` (instead of `CSWIDTH`) should be used in the environment in SunOS 5.x systems. See `chrtbl`(1M) for more information.

The file `euc.h` has the structure with fields for EUC width, screen width, and wide character width. The following functions are used to set and get EUC widths (these functions assume the environment where the `eucwidth_t` structure is needed and available):

Code Example 12-1 EUC

```
#include <eucioctl.h>         /* need others,like stropts.h*/

struct eucioc eucw;           /*for EUC_WSET/WGET to line disc*/
eucwidth_t width;             /* ret struct from _getwidth() */
/*
 * set_euc         Send EUC code widths to line discipline.
 */
set_euc(struct eucioc *e)
{
```

Code Example 12-1 EUC

```
    struct strioctl sb;

    sb.ic_cmd = EUC_WSET;
    sb.ic_timout = 15;
    sb.ic_len = sizeof(struct eucioc);
    sb.ic_dp = (char *) e;

    if (ioctl(0, I_STR, &sb) < 0)
            fail();
}
/*
 * euclook.   Get current EUC code widths from line discipline.
 */
euclook(struct eucioc *e)
{
    struct strioctl sb;

    sb.ic_cmd = EUC_WGET;
    sb.ic_timout = 15;
    sb.ic_len = sizeof(struct eucioc);
    sb.ic_dp = (char *) e;

    if (ioctl(0, I_STR, &sb) < 0)
            fail();

    printf("CSWIDTH=%d:%d,%d:%d,%d:%d",
            e->eucw[1], e->scrw[1],
            e->eucw[2], e->scrw[2],
            e->eucw[3], e->scrw[3]);
}
```

For more detailed descriptions, see *System Interfaces Guide*.

Hardware Emulation Module

If a Stream supports a terminal interface, a driver or module that understands all `ioctl`s to support terminal semantics (specified by `termio` and `termios`) is needed. If there is no hardware driver that understands all `ioctl` commands downstream from the `ldterm` module, a hardware emulation module must be placed downstream from the line discipline module. The

function of the hardware emulation module is to understand and acknowledge the `ioctls` that may be sent to the process at the Stream head and to mediate the passage of control information downstream. The combination of the line-discipline module and the hardware emulation module behaves as if there were an actual terminal on that Stream.

The hardware emulation module is necessary whenever there is no TTY driver at the end of the Stream. For example, the module is necessary in a pseudo-TTY situation where there is process- to- process communication on one system (this is discussed later in this chapter), or in a network situation where a `termio` interface is expected (for example, remote login) but there is no TTY driver on the Stream.

Most of the actions taken by the hardware emulation module are the same regardless of the underlying architecture. However, there are some actions that are different depending on whether the communication is local or remote and whether the underlying transport protocol is used to support the remote connection.

Each hardware emulation module has an open, close, read queue `put` procedure, and write queue `put` procedure.

The hardware emulation module does the following:

- Processes, if appropriate, and acknowledges receipt of the following `ioctls` on its write queue by sending an `M_IOCACK` message back upstream: `TCSETA`, `TCSETAW`, `TCSETAF`, `TCSETS`, `TCSETSW`, `TCSETSF`, `TCGETA`, `TCGETS`, and `TCSBRK`.
- Acknowledges the Extended UNIX Code (EUC) `ioctls`.
- If the environment supports windowing, it acknowledges the windowing `ioctls` `TIOCSWINSZ`, `TIOCGWINSZ`, and `JWINSIZE`. If the environment does not support windowing, an `M_IOCNAK` message is sent upstream.
- If any other `ioctls` are received on its write queue, it sends an `M_IOCNAK` message upstream. It doesn't pass any unrecognized `ioctl`'s to the slave driver.
- When the hardware emulation module receives an `M_IOCTL` message of type `TCSBRK` on its write queue, it sends an `M_IOCACK` message upstream and the appropriate message downstream. For example, an `M_BREAK` message could be sent downstream.

- When the hardware emulation module receives an M_IOCTL message on its write queue to set the baud rate to 0 (TCSETAW with CBAUD set to B0), it sends an M_IOCACK message upstream and an appropriate message downstream; for networking situations this will probably be an M_PROTO message which is a TPI T_DISCON_REQ message requesting the transport provider to disconnect.
- All other messages (M_DATA, for instance) not mentioned here are passed to the next module or driver in the Stream.

The hardware emulation module processes messages in a way consistent with the driver that exists .

STREAMS-based Pseudo-Terminal Subsystem

The STREAMS-based pseudo-terminal subsystem provides the user with an interface that is identical to the STREAMS-based terminal subsystem described earlier in this chapter. The pseudo-terminal subsystem (pseudo-TTY) supports a pair of STREAMS-based devices called the *master* device and *slave* device. The slave device provides processes with an interface that is identical to the terminal interface. However, where all devices, which provide the terminal interface, have some kind of hardware device behind them, the slave device has another process manipulating it through the master half of the pseudo terminal. Anything written on the master device is given to the slave as an input and anything written on the slave device is presented as an input on the master side.

Figure 12-2 illustrates the architecture of the STREAMS-based pseudo-terminal subsystem. The master driver called ptm is accessed through the clone driver (see clone(7)) and is the controlling part of the system. The slave driver called pts works with the line discipline module and the hardware emulation module to provide a terminal interface to the user process. An optional packetizing module called pckt is also provided. It can be pushed on the master side to support packet mode (this is discussed later).

The number of pseudo-TTY devices that can be installed on a system depends on available memory.

Line-Discipline Module

In the pseudo-TTY subsystem, the line discipline module is pushed on the slave side to present the user with the terminal interface.

`ldterm` may turn off the processing of the *c_iflag*, *c_oflag*, and *c_lflag* fields to allow processing to take place elsewhere. The `ldterm` module can also turn off all canonical processing when it receives an `M_CTL` message with the `MC_NO_CANON` command to support *remote mode*. Although `ldterm` passes through messages without processing them, the appropriate flags are set when a *get* `ioctl`, such as `TCGETA` or `TCGETS`, is issued to indicate that canonical processing is being performed.

12

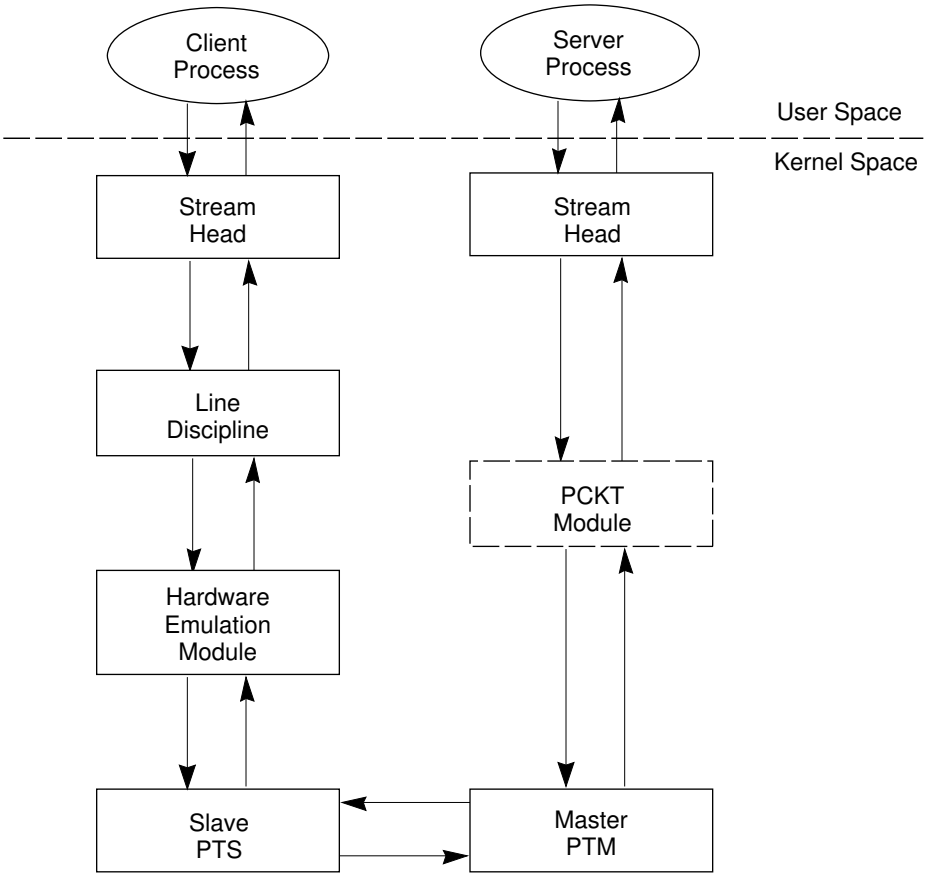

Figure 12-2 Pseudo-TTY Subsystem Architecture

Pseudo-TTY Emulation Module - PTEM

Since the pseudo-TTY subsystem has no hardware driver downstream from the `ldterm` module to process the terminal `ioctl` calls, another module that understands the `ioctl` commands is placed downstream from the `ldterm`. This module, known as `ptem`, processes all of the terminal `ioctl` commands and mediates the passage of control information downstream.

ldterm and ptem together behave like a real terminal. Since there is no real terminal or modem in the pseudo-TTY subsystem, some of the ioctl commands are ignored and cause only an acknowledgment of the command. The ptem module keeps track of the terminal parameters set by the various *set* commands such as TCSETA or TCSETAW but does not usually perform any action. For example, if one of the "set" ioctls is called, none of the bits in the *c_cflag* field of termio has any effect on the pseudo-terminal except if the baud rate is set to 0. When setting the baud rate to 0, it has the effect of hanging up the pseudo-terminal.

The pseudo-terminal has no concept of parity so none of the flags in the *c_iflag* that control the processing of parity errors have any effect. The delays specified in the *c_oflag* field are not also supported.

The ptem module does the following:

- Processes, if appropriate, and acknowledges receipt of the following ioctls on its write queue by sending an M_IOCACK message back upstream:

  TCSETA, TCSETAW, TCSETAF, TCSETS, TCSETSW, TCSETSF, TCGETA, TCGETS, and TCSBRK.

- Keeps track of the window size; information needed for the TIOCSWINSZ, TIOCGWINSZ, and JWINSIZE ioctl commands.
- When it receives any other ioctl on its write queue, it sends an M_IOCNAK message upstream.
- It passes downstream the following ioctls after processing them:

  TCSETA, TCSETAW, TCSETAF, TCSETS, TCSETSW, TCSETSF, TCSBRK, and TIOCSWINSZ.

- ptem frees any M_IOCNAK messages it receives on its read queue in case the pckt module (pckt is described in the section “Packet Mode") is not on the pseudo terminal subsystem and the above ioctls get to the master’s Stream head which would then send an M_IOCNAK message.
- In its open routine, the ptem module sends an M_SETOPTS message upstream requesting allocation of a controlling TTY.
- When the ptem module receives an M_IOCTL message of type TCSBRK on its read queue, it sends an M_IOCACK message downstream and an M_BREAK message upstream.

- When the ptem receives an ioctl message on its write queue to set the baud rate to 0 (TCSETAW with CBAUD set to B0), it sends an M_IOCACK message upstream and a zero-length message downstream.
- When it receives an M_IOCTL of type TIOCSIGNAL on its read queue, it sends an M_IOCACK downstream and an M_PCSIG upstream where the signal number is the same as in the M_IOCTL message.
- When the ptem module receives an M_IOCTL of type TIOCREMOTE on its read queue, it sends an M_IOCACK message downstream and the appropriate M_CTL message upstream to enable/disable canonical processing.
- When it receives an M_DELAY message on its read or write queue, it discards the message and does not act on it.
- When it receives an M_IOCTL message with type JWINSIZE on its write queue and if the values in the jwinsize structure of ptem are not zero, it sends an M_IOCACK message upstream with the jwinsize structure. If the values are zero, it sends an M_IOCNAK message upstream.
- When it receives an M_IOCTL message of type TIOCGWINSZ on its write queue and if the values in the winsize structure are not zero, it sends an M_IOCACK message upstream with the winsize structure. If the values are zero, it sends an M_IOCNAK message upstream. It also saves the information passed to it in the winsize structure and sends a STREAMS signal message for signal SIGWINCH upstream to the slave process if the size changed.
- When the ptem module receives an M_IOCTL message with type TIOCGWINSZ on its read queue and if the values in the winsize structure are not zero, it sends an M_IOCACK message downstream with the winsize structure. If the values are zero, it sends an M_IOCNAK message downstream. It also saves the information passed to it in the winsize structure and sends a STREAMS signal message for signal SIGWINCH upstream to the slave process if the size changed.
- All other messages not mentioned above are passed to the next module or driver.

Data Structure

SunOS 5.3 reserves the right to change ptem's internal implementation. This structure should be relevant only to people wanting to change the module.

Each instantiation of the ptem module is associated with a local area. These data is held in a structure called ptem that has the following format:

```
struct ptem
{
    long cflags;              /* copy of c_flags */
    mblk_t *dack_ptr;         /* pointer to preallocated msg blk
                                 used to send disconnect */
    queue_t *q_ptr;           /* pointer to ptem's read queue */
    struct winsize wsz;/*struct to hold windowing info*/
    unsigned short state; /* state of ptem entry */
};
```

When the ptem module is pushed onto the slave side Stream, a search of the ptem structure is made for a free entry (*state* is not set to INUSE). The *c_cflag*s of the termio structure and the windowing variables are stored in *cflags* and *wsz* respectively. The *dack_ptr* is a pointer to a message block used to send a zero-length message whenever a hang-up occurs on the slave side.

Open and Close Routines

In the open routine of ptem a STREAMS message block is allocated for a zero-length message for delivering a hang-up message; this allocation of a buffer is done before it is needed to ensure that a buffer is available. An M_SETOPTS message is sent upstream to set the read side Stream head queues, to assign high and low water marks (1024 and 256 respectively), and to establish a controlling terminal.

The same default values as for the line discipline module are assigned to *cflags*,and INUSE to the *state* field.

Note – These default values are currently being examined and may change in the future.

The open routine fails if:

- No free entries are found when the ptem structure is searched.
- *sflag* is not set to MODOPEN.
- A zero-length message can not be allocated (no buffer is available).

- A `stroptions` structure can not be allocated.

The close routine is called on the last close of the slave side Stream. Pointers to read and write queue are cleared and the buffer for the zero-length message is freed.

Remote Mode

A feature known as *remote mode* is available with the pseudo-TTY subsystem. This feature is used for applications that perform the canonical function normally done by the `ldterm` module and TTY driver. The remote mode allows applications on the master side to turn off the canonical processing. An `ioctl TIOCREMOTE` with a nonzero parameter (`ioctl(fd, TIOCREMOTE, 1)`) is issued on the master side to enter the remote mode. When this occurs, an `M_CTL` message with the command `MC_NO_CANON` is sent to the `ldterm` module indicating that data should be passed when received on the read side and no canonical processing is to take place. The remote mode may be disabled by:

```
ioctl(fd, TIOCREMOTE, 0).
```

Packet Mode

The STREAMS-based pseudo-terminal subsystem also supports a feature called *packet mode*. This is used to inform the process on the master side when *state* changes have occurred in the pseudo-TTY. Packet mode is enabled by pushing the `pckt` module on the master side. Data written on the master side is processed normally. When data is written on the slave side or when other messages are encountered by the `pckt` module, a header is added to the message so it can be subsequently retrieved by the master side with a `getmsg` operation.

The `pckt` module does the following:

- When a message is passed to this module on its write queue, the module does no processing and passes the message to the next module or driver.
- The `pckt` module creates an `M_PROTO` message when one of the following messages is passed to it:

  `M_DATA`, `M_IOCTL`, `M_PROTO/M_PCPROTO`, `M_FLUSH`, `M_START/M_STOP`, `M_STARTI/M_STOPI`, and `M_READ`.

All other messages are passed through. The M_PROTO message is passed upstream and retrieved when the user issues getmsg(2).

- If the message is an M_FLUSH message, pckt does the following:

  If the flag is FLUSHW, it is changed to FLUSHR (because FLUSHR was the original flag before the pts driver changed it), packetized into an M_PROTO message, and passed upstream. To prevent the Stream head's read queue from being flushed, the original M_FLUSH message must not be passed upstream.

  If the flag is FLUSHR, it is changed to FLUSHW, packetized into an M_PROTO message, and passed upstream. In order to flush of the write queues properly, an M_FLUSH message with the FLUSHW flag set is also sent upstream.

  If the flag is FLUSHRW, the message with both flags set is packetized and passed upstream. An M_FLUSH message with the FLUSHW flag set is also sent upstream.

Pseudo-TTY Drivers - ptm and pts

To use the pseudo-TTY subsystem, a node for the master side driver /dev/ptmx and N number of slave drivers (N is determined at installation time) must be installed. The names of the slave devices are /dev/pts/M where M has the values 0 through N-1. A user accesses a pseudo-TTY device through the master device (called ptm) that in turn is accessed through the clone driver (see clone(7)). The master device is set up as a clone device where its major device number is the major for the clone device and its minor device number is the major for the ptm driver.

The master pseudo driver is opened via the open(2) system call with /dev/ptmx as the device to be opened. The clone open finds the next available minor device for that major device; a master device is available only if it and its corresponding slave device are not already open. There are no nodes in the file system for master devices.

When the master device is opened, the corresponding slave device is automatically locked out. No user may open that slave device until it is unlocked. A user may invoke a function grantpt that will change the owner of the slave device to that of the user who is running this process, change the group id to TTY, and change the mode of the device to *0620*. Once the

permissions have been changed, the device may be unlocked by the user. Only the owner or super-user can access the slave device. The user must then invoke the `unlockpt` function to unlock the slave device. Before opening the slave device, the user must call the `ptsname` function to obtain the name of the slave device. The functions `grantpt`, `unlockpt`, and `ptsname` are called with the file descriptor of the master device. The user may then invoke the `open` system call with the name that was returned by the `ptsname` function to open the slave device.

The following example shows how a user may invoke the pseudo-TTY subsystem:

```
int fdm fds;
char *slavename;
extern char *ptsname();

fdm = open("/dev/ptmx", O_RDWR);      /* open master */
grantpt(fdm);                  /* change permission of slave */
unlockpt(fdm);                        /* unlock slave */
slavename = ptsname(fdm);             /* get name of slave */
fds = open(slavename, O_RDWR);/* open slave */
ioctl(fds, I_PUSH, "ptem");           /* push ptem */
ioctl(fds, I_PUSH, "ldterm");/* push ldterm */
```

Unrelated processes may open the pseudo device. The initial user may pass the master file descriptor using a STREAMS-based pipe or a slave name to another process to enable it to open the slave. After the slave device is open, the owner is free to change the permissions.

Note – Certain programs such as `write` and `wall` are set group-id (setgid) to TTY and are also able to access the slave device.

After both the master and slave have been opened, the user has two file descriptors that provide full-duplex communication using two Streams. The two Streams are automatically connected. The user may then push modules onto either side of the Stream. The user also needs to push the `ptem` and `ldterm` modules onto the slave side of the pseudo-terminal subsystem to get terminal semantics.

The master and slave drivers pass all STREAMS messages to their adjacent queues. Only the M_FLUSH needs some processing. Because the read queue of one side is connected to the write queue of the other, the FLUSHR flag is changed to FLUSHW flag and vice versa.

When the master device is closed, an M_HANGUP message is sent to the slave device which will render the device unusable. The process on the slave side gets the *errno* ENXIO when attempting to write on that Stream but it will be able to read any data remaining on the Stream head read queue. When all the data has been read, read returns 0 indicating that the Stream can no longer be used.

On the last close of the slave device, a zero-length message is sent to the master device. When the application on the master side issues a read or getmsg and 0 is returned, the user of the master device decides whether to issue a close that dismantles the pseudo-terminal subsystem. If the master device is not closed, the pseudo-TTY subsystem will be available to another user to open the slave device.

Since zero-length messages are used to indicate that the process on the slave side has closed and should be interpreted that way by the process on the master side, applications on the slave side should not write zero-length messages. If that occurs, the write returns 0, and the zero-length message is discarded by the ptem module.

The standard STREAMS system calls can access the pseudo-TTY devices. The slave devices support the O_NDELAY and O_NONBLOCK flags. Since the master side does not act like the terminal, if O_NONBLOCK or O_NDELAY is set, read on the master side returns -1 with *errno* set to EAGAIN if no data is available, and write returns -1 with *errno* set to EAGAIN if there is internal flow control.

The master driver supports the ISPTM and UNLKPT ioctls that are used by the functions grantpt, unlockpt, and ptsname (see grantpt(3C), unlockpt(3C), ptsname(3C)). The ioctl ISPTM determines whether the file descriptor is that of an open master device. On success, it returns the major/minor number (type dev_t) of the master device which can be used to determine the name of the corresponding slave device. The ioctl UNLKPT unlocks the master and slave devices. It returns 0 on success. On failure, the *errno* is set to EINVAL indicating that the master device is not open.

The format of these commands is:

```
int ioctl (int fd, int command, int arg)
```

where `command` is either `ISPTM` or `UNLKPT` and `arg` is 0. On failure, -1 is returned.

When data is written to the master side, the entire block of data written is treated as a single line. The slave side process reading the terminal receives the entire block of data. Data is not input edited by the `ldterm` module regardless of the terminal mode. The master side application is responsible for detecting an interrupt character and sending an interrupt signal `SIGINT` to the process in the slave side. This can be done as follows:

```
    ioctl (fd, TIOCSIGNAL, SIGINT)
```

where `SIGINT` is defined in the file `<signal.h>`. When a process on the master side issues this `ioctl`, the argument is the number of the signal that should be sent. The specified signal is then sent to the process group on the slave side.

To summarize, the master driver and slave driver have the following characteristics:

- Each master driver has one-to-one relationship with a slave device based on major/minor device numbers.
- Only one open is allowed on a master device. Multiple opens are allowed on the slave device according to standard file mode and ownership permissions.
- Each slave driver minor device has a node in the file system.
- An open on a master device automatically locks out an open on the corresponding slave driver.
- A slave cannot be opened unless the corresponding master is open and has unlocked the slave.
- To provide a TTY interface to the user, the `ldterm` and `ptem` modules are pushed on the slave side.
- A close on the master sends a hang-up to the slave and renders both Streams unusable after all data have been consumed by the process on the slave side.
- The last close on the slave side sends a zero-length message to the master but does not sever the connection between the master and slave drivers.

grantpt

The `grantpt` function changes the mode and the ownership of the slave device that is associated with the given master device. Given a file descriptor *fd*, `grantpt` first checks that the file descriptor is that of the master device. If so, it obtains the name of the associated slave device and sets the user id to that of the user running the process and the group id to TTY. The mode of the slave device is set to *0620*.

If the process is already running as root, the permission of the slave can be changed directly without invoking this function. The interface is:

```
grantpt (int fd);
```

The `grantpt` function returns 0 on success and -1 on failure. It fails if one or more of the following occurs: `fd` is not an open file descriptor, `fd` is not associated with a master device, the corresponding slave could not be accessed, or a system call failed because no more processes could be created.

unlockpt

The `unlockpt` function clears a lock flag associated with a master/slave device pair. Its interface is:

```
unlockpt (int fd);
```

The `unlockpt` returns 0 on success and -1 on failure. It fails if one or more of the following occurs: `fd` is not an open file descriptor or `fd` is not associated with a master device.

ptsname

The `ptsname` function returns the name of the slave device that is associated with the given master device. It first checks that the file descriptor is that of the master. If it is, it then determines the name of the corresponding slave device `/dev/pts/M` and returns a pointer to a string containing the null-terminated path name. The return value points to static data whose content is overwritten by each call. The interface is:

```
char *ptsname (int fd);
```

The `ptsname` function returns a non-`NULL` path name upon success and a `NULL` pointer upon failure. It fails if one or more of the following occurs: `fd` is not an open file descriptor or `fd` is not associated with the master device.