Line Disciplines

Line disciplines provide an elegant mechanism that lets you use the same serial driver to run different technologies. The low-level physical driver and the tty driver handle the transfer of data to and from the hardware, while line disciplines are responsible for processing the data and transferring it between kernel space and user space.

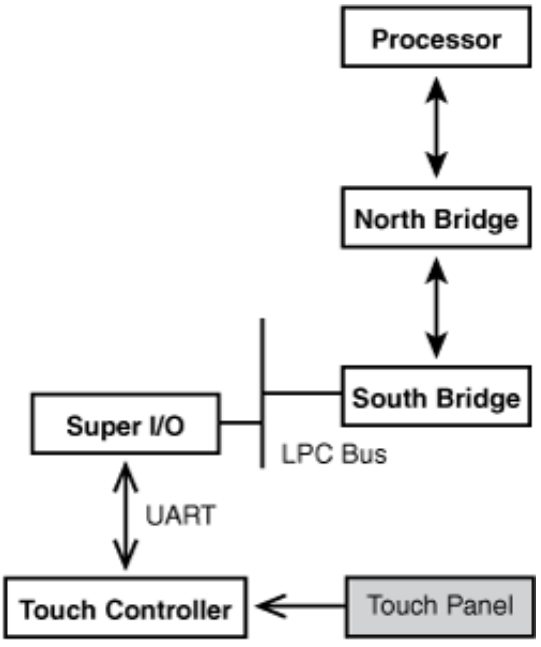
The serial subsystem supports 17 standard line disciplines. The default line discipline that gets attached when you open a serial port is `N_TTY`, which implements terminal I/O processing. `N_TTY` is responsible for "cooking" characters received from the keyboard. Depending on user request, it maps the control character to newline, converts lowercase to uppercase, expands tabs, and echoes characters to the associated VT. `N_TTY` also supports a *raw* mode used by editors, which leaves all the preceding processing to user applications. See Figure 7.3 in the next chapter, "Input Drivers," to learn how the keyboard subsystem is connected to `N_TTY`. The example tty drivers listed at the end of the previous section, "TTY Drivers," use `N_TTY` by default.

Line disciplines also implement network interfaces over serial transport protocols. For example, line disciplines that are part of the *Point-to-Point Protocol* (`N_PPP`) and the *Serial Line Internet Protocol* (`N_SLIP`) subsystems, frame packets, allocate and populate associated networking data structures, and pass the data on to the corresponding network protocol stack. Other line disciplines handle *Infrared Data* (`N_IRDA`) and the *Bluetooth Host Control Interface* (`N_HCI`).

Device Example: Touch Controller

Let's take a look at the internals of a line discipline by implementing a simple line discipline for a serial touch-screen controller. Figure 6.6 shows how the touch controller is connected on an embedded laptop derivative. The *Finite State Machine* (FSM) of the touch controller is a candidate for being implemented as a line discipline because it can leverage the facilities and interfaces offered by the serial layer.

Figure 6.6. Connection diagram of a touch controller on a PC-derivative.



Open and Close

To create a line discipline, define a `struct tty_ldisc` and register a prescribed set of entry points with the kernel. Listing 6.2 contains a code snippet that performs both these operations for the example touch controller.

Listing 6.2. Line Discipline Operations

Code View:

```
struct tty_ldisc n_touch_ldisc = {
  TTY_LDISC_MAGIC,          /* Magic */
  "n_tch",                  /* Name of the line discipline */
  N_TCH,                    /* Line discipline ID number */
  n_touch_open,             /* Open the line discipline */
  n_touch_close,            /* Close the line discipline */
  n_touch_flush_buffer,     /* Flush the line discipline's read
                               buffer */
  n_touch_chars_in_buffer, /* Get the number of processed characters in
                               the line discipline's read buffer */
  n_touch_read,             /* Called when data is requested
                               from user space */
  n_touch_write,            /* Write method */
  n_touch_ioctl,            /* I/O Control commands */
  NULL,                     /* We don't have a set_termios
                               routine */
  n_touch_poll,             /* Poll */
  n_touch_receive_buf,      /* Called by the low-level driver
                               to pass data to user space*/
  n_touch_receive_room,     /* Returns the room left in the line
                               discipline's read buffer */
  n_touch_write_wakeup      /* Called when the low-level device
                               driver is ready to transmit more
                               data */
};

/* ... */

if ((err = tty_register_ldisc(N_TCH, &n_touch_ldisc))) {
  return err;
}
```

In Listing 6.2, `n_tch` is the name of the line discipline, and `N_TCH` is the line discipline identifier number. You have to specify the value of `N_TCH` in *include/linux/tty.h*, the header file that contains all line discipline definitions. Line disciplines active on your system can be found in */proc/tty/ldiscs*.

Line disciplines gather data from their half of the tty flip buffer, process it, and copy the resulting data to a local *read* buffer. For `N_TCH`, `n_touch_receive_room()` returns the memory left in the read buffer, while `n_touch_chars_in_buffer()` returns the number of processed characters in the read buffer that are ready to be delivered to user space. `n_touch_write()` and `n_touch_write_wakeup()` do nothing because `N_TCH` is a read-only device. `n_touch_open()` takes care of allocating memory for the main line discipline data structures, as shown in Listing 6.3.

Listing 6.3. Opening the Line Discipline

Code View:

```
/* Private structure used to implement the Finite State Machine
(FSM) for the touch controller. The controller and the processor
communicate using a specific protocol that the FSM implements */
struct n_touch {
  int current_state;        /* Finite State Machine */
  spinlock_t touch_lock;    /* Spinlock */
  struct tty_struct *tty;   /* Associated tty */
  /* Statistics and other housekeeping */
  /* ... */
} *n_tch;


/* Device open() */
static int
n_touch_open(struct tty_struct *tty)
{
  /* Allocate memory for n_tch */
  if (!(n_tch = kmalloc(sizeof(struct n_touch), GFP_KERNEL))) {
    return -ENOMEM;
  }
  memset(n_tch, 0, sizeof(struct n_touch));

  tty->disc_data = n_tch; /* Other entry points now
                             have direct access to n_tch */
  /* Allocate the line discipline's local read buffer
     used for copying data out of the tty flip buffer */
```

```
  tty->read_buf = kmalloc(BUFFER_SIZE, GFP_KERNEL);
  if (!tty->read_buf) return -ENOMEM;

  /* Clear the read buffer */
  memset(tty->read_buf, 0, BUFFER_SIZE);

  /* Initialize lock */
  spin_lock_init(&ntch->touch_lock);

  /* Initialize other necessary tty fields.
     See drivers/char/n_tty.c for an example */
  /* ... */

  return 0;
}
```

You might want to set `N_TCH` as the default line discipline (rather than `N_TTY`) when-ever the serial port connected to the touch controller is opened. See the section "Changing Line Disciplines" to see how to change line disciplines from user space.

Read Path

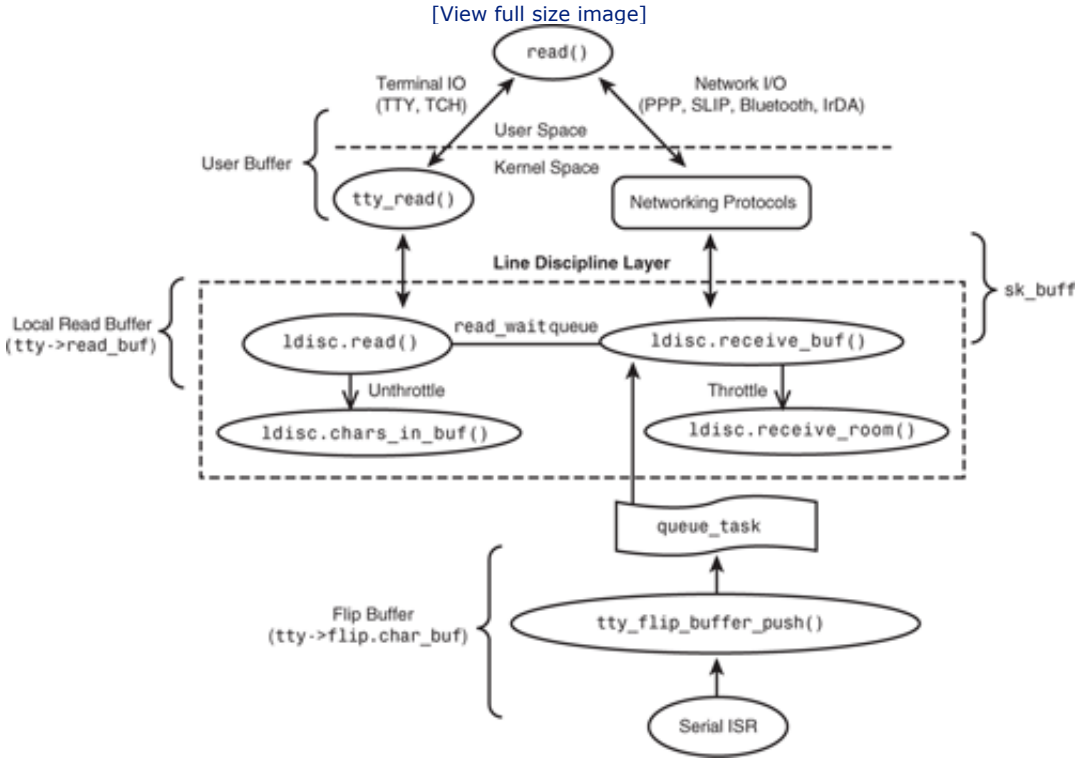
For interrupt-driven devices, the read data path usually consists of two threads working in tandem:

1. A top thread originating from the user process requesting the read
2. A bottom thread springing from the interrupt service routine that collects data from the device

Figure 6.7 shows these threads associated with the read data flow. The interrupt handler queues the `receive_buf()` method (`n_touch_receive_buf()` in our example) as a task. You can override this behavior by setting the `tty->low_latency` flag.

Figure 6.7. Line discipline read path.

[View full size image]



The touch controller and the processor communicate using a specific protocol described in the controller's data sheet. The driver implements this communication protocol using an FSM as indicated earlier. Listing 6.4 encodes this FSM as part of the `receive_buf()` entry point, `n_touch_receive_buf()`.

Listing 6.4. The `n_touch_receive_buf()` Method

Code View:

```
static void
n_touch_receive_buf(struct tty_struct *tty,
                    const unsigned char *cp, char *fp, int count)
{

  /* Work on the data in the line discipline's half of
     the flip buffer pointed to by cp */
  /* ... */

  /* Implement the Finite State Machine to interpret commands/data
     arriving from the touch controller and put the processed data
     into the local read buffer */
.....................................................................................
  /* Datasheet-dependent Code Region */
  switch (tty->disc_data->current_state) {
    case RESET:
      /* Issue a reset command to the controller */
      tty->driver->write(tty, 0, mode_stream_command,
                         sizeof(mode_stream_command));
      tty->disc_data->current_state = STREAM_DATA;
      /* ... */
      break;
    case STREAM_DATA:
      /* ... */
      break;
    case PARSING:
      /* ... */
      tty->disc_data->current_state = PARSED;
      break;
    case PARSED:
      /* ... */
    }
.....................................................................................

  if (tty->disc_data->current_state == PARSED) {
    /* If you have a parsed packet, copy the collected coordinate
       and direction information into the local read buffer */
    spin_lock_irqsave(&tty->disc_data->touch_lock, flags);
    for (i=0; i < PACKET_SIZE; i++) {
      tty->disc_data->read_buf[tty->disc_data->read_head] =
                          tty->disc_data->current_pkt[i];
      tty->disc_data->read_head =
                (tty->disc_data->read_head + 1) & (BUFFER_SIZE - 1);
      tty->disc_data->read_cnt++;
    }
    spin_lock_irqrestore(&tty->disc_data->touch_lock, flags);

    /* ... */ /* See Listing 6.5 */

  }
}
```

`n_touch_receive_buf()` processes the data arriving from the serial driver. It exchanges a series of commands and responses with the touch controller and puts the received coordinate and direction (press/release) information into the line discipline's read buffer. Accesses to the read buffer have to be serialized using a spinlock because it's used by both `ldisc.receive_buf()` and `ldisc.read()` threads shown in Figure 6.7 (`n_touch_receive_buf()` and `n_touch_read()`, respectively, in our example). As you can see in Listing 6.4, `n_touch_receive_buf()` dispatches commands to the touch controller by directly calling the `write()` entry point of the serial driver.

`n_touch_receive_buf()` needs to do a couple more things:

1. The top `read()` thread in Figure 6.7 puts the calling process to sleep if no data is available. So, `n_touch_receive_buf()` has to wake up that thread and let it read the data that was just processed.

2. If the line discipline is running out of read buffer space, `n_touch_receive_buf()` has to request the

serial driver to throttle data arriving from the device. `ldisc.read()` is responsible for requesting the corresponding unthrottling when it ferries the data to user space and frees memory in the read buffer. The serial driver uses software or hardware flow control mechanisms to achieve the throttling and unthrottling.

Listing 6.5 performs these two operations.

Listing 6.5. Waking Up the Read Thread and Throttling the Serial Driver

```
/* n_touch_receive_buf() continued.. */

/* Wake up any threads waiting for data */
if (waitqueue_active(&tty->read_wait) &&
    (tty->read_cnt >= tty->minimum_to_wake))
  wake_up_interruptible(&tty->read_wait);
}
/* If we are running out of buffer space, request the
   serial driver to throttle incoming data */
if (n_touch_receive_room(tty) < TOUCH_THROTTLE_THRESHOLD) {
  tty->driver.throttle(tty);
}
/* ... */
```

A wait queue (`tty->read_wait`) is used to synchronize between the `ldisc.read()` and `ldisc.receive_buf()` threads. `ldisc.read()` adds the calling process to the wait queue when it does not find data to read, while `ldisc.receive_buf()` wakes the `ldisc.read()` thread when there is data available to be read. So, `n_touch_read()` does the following:

- If there is no data to be read yet, put the calling process to sleep on the `read_wait` queue. The process gets woken by `n_touch_receive_buf()` when data arrives.
- If data is available, collect it from the local read buffer (`tty->read_buf[tty->read_tail]`) and dispatch it to user space.
- If the serial driver has been throttled and if enough space is available in the read buffer after this read, ask the serial driver to unthrottle.

Networking line disciplines usually allocate an `sk_buff` (the basic Linux networking data structure discussed in Chapter 15, "Network Interface Cards") and use this as the read buffer. They don't have a `read()` method, because the corresponding `receive_buf()` copies received data into the allocated `sk_buff` and directly passes it to the associated protocol stack.

Write Path

A line discipline's `write()` entry point performs any post processing that is required before passing the data down to the low-level driver.

If the underlying driver is not able to accept all the data that the line discipline offers, the line discipline puts the requesting thread to sleep. The driver's interrupt handler wakes the line discipline when it is ready to receive more data. To do this, the driver calls the `write_wakeup()` method registered by the line discipline. The associated synchronization is done using a wait queue (`tty->write_wait`), and the operation is similar to that of the `read_wait` queue described in the previous section.

Many networking line disciplines have no `write()` methods. The protocol implementation directly transmits the frames down to the serial device driver. However, these line disciplines usually still have a `write_wakeup()` entry point to respond to the serial driver's request for more transmit data.

`N_TCH` does not need a `write()` method either, because the touch controller is a read-only device. As you saw in Listing 6.4, routines in the receive path directly talk to the low-level UART driver when they need to send command frames to the controller.

I/O Control

A user program can send commands to a device via `ioctl()` calls, as discussed in Chapter 5, "Character Drivers." When an application opens a serial device, it can usually issue three classes of ioctls to it:

- Commands supported by the serial device driver, such as `TIOCMSET` that sets modem information
- Commands supported by the tty driver, such as `TIOCSETD` that changes the attached line discipline
- Commands supported by the attached line discipline, such as a command to reset the touch controller in the case of `N_TCH`

The `ioctl()` implementation for `N_TCH` is largely standard. Supported commands depend on the protocol described in the touch controller's data sheet.

More Operations

Another line discipline operation is `flush_buffer()`, which is used to flush any data pending in the read buffer. `flush_buffer()` is also called when a line discipline is closed. It wakes up any read threads that are waiting for more data as follows:

```
if (tty->link->packet){
  wake_up_interruptible(&tty->disc_data->read_wait);
}
```

Yet another entry point (not supported by `N_TCH`) is `set_termios()`. The `N_TTY` line discipline supports the `set_termios()` interface, which is used to set options specific to line discipline data processing. For example, you may use `set_termios()` to put the line discipline in raw mode or "cooked" mode. Some options specific to the touch controller (such as changing the baud rate, parity, and number of stop bits) are implemented by the `set_termios()` method of the underlying device driver.

The remaining entry points such as `poll()` are fairly standard, and you can return to Chapter 5 in case you need assistance.

You may compile your line discipline as part of the kernel or dynamically load it as a module. If you choose to compile it as a module, you must, of course, also provide functions to be called during module initialization and exit. The former is usually the same as the `init()` method. The latter needs to clean up private data structures and unregister the line discipline. Unregistering the discipline is a one-liner:

```
tty_unregister_ldisc(N_TCH);
```

An easier way to drive a serial touch controller is by leveraging the services offered by the kernel's *input* subsystem and the built-in *serport* line discipline. We look at that technique in the next chapter.

Changing Line Disciplines

`N_TCH` gets bound to the low-level serial driver when a user space program opens the serial port connected to the touch controller. But sometimes, a user-space application might want to attach a different line discipline to the device. For instance, you might want to write a program that dumps raw data received from the touch controller without processing it. Listing 6.6 opens the touch controller and changes the line discipline to `N_TTY` to dump the data that is coming in.

Listing 6.6. Changing a Line Discipline from User Space

```
fd = open("/dev/ttySX", O_RDONLY | O_NOCTTY);

/* At this point, N_TCH is attached to /dev/ttySX, the serial port used
   by the touch controller. Switch to N_TTY */
ldisc = N_TTY;
ioctl(fd, TIOCSETD, &ldisc);

/* Set termios to raw mode and dump the data coming in */
/* ... */
```

The `TIOCSETD ioctl()` closes the current line discipline and opens the newly requested line discipline.