Writing a Kernel Line Discipline

Look at the internals of a line discipline and implement a simple line discipline for a serial touch screen controller.

By Sreekrishnan Venkateswaran
Tuesday, March 15th, 2005

From June 1999 to February 2001, Linux Magazine' s "Gearheads" column presented a variety of articles about Linux kernel and device driver development. This month, "Gearheads" returns as a monthly column, reflecting an increasing interest in porting and adapting the Linux kernel to hardware of all kinds. If you'd like to suggest topics for "Gearheads," please see the contact information at the end of the article.
Line disciplines are an elegant way to use the same serial device driver to run different technologies. Terminals and many other devices, including equipment based on Point-to-Point Protocol (PPP), Serial Line Internet Protocol (SLIP), Bluetooth, and Infrared Data (IrDa), make use of a serial transport.
Line disciplines reside one layer above the serial driver and shape its behavior. The serial drivers handle the transfer of data to and from the hardware, while the line disciplines are responsible for processing the data and transferring it between kernel space and user space.
Line disciplines are also used to implement network interfaces over serial transport protocols. For example, the PPP, SLIP, and Bluetooth line disciplines perform packet framing, allocate and populate corresponding networking data structures, and pass the data on to the corresponding protocol layers.
There are sixteen standard line disciplines supported by the Linux kernel. The default line discipline that gets attached when a serial port gets opened is the N_TTY line discipline, which implements terminal I/O processing.
In this column, let's look at the internals of a line discipline and implement a simple line discipline for a serial touch screen controller.

Line Discipline Operations

To create a line discipline, it must be defined using struct tty_ldisc and its entry points must be registered with the kernel. Listing One shows a code snippet that performs both operations.
LISTING ONE: Line Discipline Operations

```
struct tty_ldisc n_touch_ldisc = {
  TTY_LDISC_MAGIC, /* Magic */
  "n_tch", /* Name of our Line Discipline */
  N_TCH, /* Our Line Discipline ID number */
  n_touch_open, /* Called on open */
  n_touch_close, /* Called on close */
  n_touch_flush_buffer, /* Flush the line discipline read buffer */
  n_touch_chars_in_buffer, /* Called to get the number of processed
                             characters in the line discipline buffer */
  n_touch_read, /* Called when data is requested from user space */
  n_touch_write, /* Called to write data */
  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 from the serial driver to pass on data */
  n_touch_receive_room, /* Returns the room left in the line
  discipline read buffer */
  n_touch_write_wakeup  /* Called when the serial device driver is
  ready to transmit more data */
 };

…

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

In the code, "n_tch" is the name of our line discipline, while N_TCH is the line discipline identifier number. (N_TCH must be added to include/asm/termios.h, where line disciplines are defined. All of the active line disciplines in your system can be found in the process file system entry /proc/tty/ldiscs.). Because the touch controller is a read-only device, you don't really need the n_touch_write() and n_touch_write_wakeup() entry points, but leave them for now to understand the data flow in the write path.
All of the states associated with serial data transfer are kept in a tty structure defined in include/linux/tty.h.
The data buffer used by the serial driver is called a flip buffer (tty->flip.char_buf). The serial driver uses one half of the flip buffer for gathering data, while the line discipline uses the other half for

processing the data. The buffer pointers used by the serial driver and the line discipline are then flipped, and the process continues. (Have a look at the function flush_to_ldisc() in tty_io.c to see the flip.)
The line discipline gathers the data from its current half of the flip buffer, processes it, and copies the “cooked” data to a local read buffer. Networking line disciplines usually allocate an skbuff (which is the basic Linux networking data structure) and use this as the local read buffer. The n_touch_receive_room() entry point for the line discipline returns the memory left in its read buffer, while the n_touch_chars_in_buffer() entry point returns the number of buffered processed characters that are ready to be delivered to user space.
The line discipline n_touch_open() entry point takes care of allocating the memory for the main line discipline data structures, as shown in Listing Two.
LISTING TWO: An example of a line discipline open() entry point

```
static int n_touch_open(struct tty_struct *tty)
{
 /* Our private data structure to implement the Finite
  * State Machine for the touch controller
  */
struct n_touch n_tch;

/* Allocate memory for our private data structure used to implement
 * the Finite State Machine for the touch controller
 */
if (!(n_tch = kmalloc (sizeof (struct n_touch), GFP_KERNEL))) {
  return -ENFILE;
}
memset (n_tch, 0, sizeof (struct n_touch));

/* Every entry point will now have access to our private data structure */
tty->disc_data = n_tch;

/ * Allocate the line discipline’s local read buffer that we
  * use for copying data out of the tty flip buffer
  */
tty->disc_data->read_buf = kmalloc (BUFFER_SIZE, GFP_KERNEL);
if (!tty->disc_data->read_buf)
  return -ENOMEM;
memset (tty->disc_data->read_buf, 0, BUFFER_SIZE);

/* Initialize other necessary fields in the tty data structures
 * (see n_tty.c for an example)
 */

/* ...  */
return 0;
}
```

You might also want to modify the tty driver (tty_io.c) to set N_TCH as the default line discipline whenever 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.)

The Line Discipline Read Path

For interrupt driven devices, the read data path usually consists of at least two threads working in tandem: a top thread originating from the user process requesting the read and a bottom thread originating from the interrupt service routine that grabs the data from the device.
Figure One shows the threads associated with the read data flow and the corresponding data structures that are used. You can see the user initiated read thread getting data from the line discipline buffers, as well as the serial driver initiated bottom thread supplying data to a line discipline entry point. The interrupt handler queues the line discipline receive_buf() entry point (named n_touch_receive_buf() in the touch screen code) as a task, unless low latency has been requested using the tty->low_latency flag).
FIGURE ONE: The read data path for a line discipline

Listing Three is a code snippet from the n_touch_receive_buf() function.
LISTING THREE: The n_touch_receive_buf() operation

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

/* Work on the data in our half of the flip buffer pointed to by cp */

/* ... */

/* Implement the Finite State Machine to interpret commands and data from the
 * touch controller and put the processed data into our local read buffer
 */
switch (tty->disc_data->current_state) {
  /* We had previously received a Reset ACK from the controller */
  case RESET:
    / * Ask the controller to go into data stream mode */
    tty->driver.write (tty, 0, mode_stream_command,
      sizeof (mode_stream_command));
    tty->disc_data->current_state = STREAM_DATA;
    /* ... */

  case PARSING:
    /* Collect coordinate, direction information into our data structure */
    /* ... */

  case PARSED:
    /* Put the co-ordinate and direction information into the read buffer */
    spin_lock_irqsave (&tty->disc_data->read_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->read_lock, flags);
   /* ... */
 }
```

The n_touch_receive_buf() entry point cooks the data that is being received from the serial driver. Here, it exchanges a series of commands and responses with the touch controller and puts the touch screen coordinate and direction (press/release) data received from the controller into the line discipline read buffer in an appropriate format. (You “talk” to the touch screen controller by directly calling the write entry point of the serial device driver.)
The finite state machine implementation depends on the particular touch screen controller that is used, and is written according to the protocol described in the data sheet. Also, references to the read data buffer have to be serialized using a spin lock since it’s accessed by both the ldisc.receive_buf() and the ldisc.read() threads shown in Figure One (again, n_touch_receive_buf() and n_touch_read(), respectively, in the sample code).
A couple more things need to be done.
1.The top read() thread that you see in Figure One has to put the calling process to sleep if no data is available. So, you need to wake up that thread and let it read the data that was just processed.
2.If read buffer space is running out, you need to inform the serial driver and request it to throttle data that is coming from the device. The corresponding unthrottling is done by the ldisc.read() entry point when it copies the data to user space and frees up room in the buffer. The serial driver uses software or hardware flow control mechanisms to do the throttling and unthrottling. These two operations are shown in Listing Four.
LISTING FOUR: Waking up the read thread and throttling the serial driver

```
 …

 /* n_touch_receive_buf() continued */

 if (waitqueue_active (&tty->disc_data->read_wait) &&
   (tty->disc_data->read_cnt >= tty->minimum_to_wake))
   wake_up_interruptable (&tty->disc_data->read_wait);

 /* If we are running out of buffer space, request the serial
   * driver to throttle incoming data
   */
 if (n_touch_receive_room (tty) < THRESHOLD_THROTTLE) {
   tty->driver.throttle (tty);
 }

 /* ... */
```

A wait queue (tty->disc_data->read_wait) synchronizes between the ldisc.read() and ldisc.receive_buf() threads. The ldisc.read() thread adds the calling process to the wait queue when it finds no data to read, while ldisc.receive_buf() wakes the read thread waiting on the queue when there is data available to be read.
The read entry point of the new line discipline, named n_touch_read() here, also needs to do the following:
1.If there’s no data to be read yet, put the calling process to sleep on the read_wait queue. The process will get woken by ldisc.receive_buf() when data arrives.
2.If data is available, grab it from the local read buffer (tty->disc_data->read_buf[tty->disc_data->read_tail]) and copy the data to user space.
3.If the serial driver has been throttled and if enough space has been freed in the read buffer after this read, unthrottle the serial driver.
Networking line disciplines usually do not have the read entry point, since the corresponding receive_buf entry point copies the received data into allocated skbuffs, and directly passes it on to the corresponding protocol layers.

Line Discipline Transmit Path

A line discipline’s write entry point performs any post data processing that is required and passes the data on to the serial driver.
If the serial driver is not able to accept all the data that the line discipline write entry point has to offer, the line discipline puts the calling process to sleep. The serial driver interrupt handler wakes the line discipline when it is ready to receive more data. To do this, the driver calls the write_wakeup() entry point registered by the line discipline. This synchronization is done using a wait queue (tty->disc_data->write_wait), and the operation is similar to that described for the read_wait queue in the previous section.
Many networking line disciplines (like PPP) have no write entry points. The protocol code 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.
The touch screen’s line discipline doesn’t have a write entry point since the touch controller is a read-only device and the receive path routines talk directly to the serial driver when they need to send any command frames to the controller.

Line Discipline ioctl()

A user program can send commands to a device via I/O control (ioctl()) calls. When a program opens a serial device, it can usually issue three kinds of ioctl() commands to it:
*Commands supported by the serial device driver, such as the TIOCMSET command to set modem information.
*Commands supported by the attached line discipline, such as a command to reset the touch controller.
*Commands supported by the kernel tty layer, including the TIOCSETD command to change the

attached line discipline.
The ioctl() implementation for the new line discipline is largely standard, and the commands supported in the ioctl() function depends on the protocol described in the data sheet for the touch screen device.

Other Line Discipline Operations

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

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

Yet another entry point (not supported in this sample line discipline) 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, such as whether the user wants to put the line discipline in raw mode or cooked mode.
Some of the set_termios() options needed to support the touch controller (like changing the baud rate, parity, and number of stop bits) is implemented by the set_termios() operation of the serial device driver.
The remaining entry points like poll are fairly standard in Linux.
You can 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 also provide functions that will be called during module initialization and exit. The former is just a wrapper to your init() function, while the latter needs to clean up private data structures and unregister the line discipline. Unregistering the discipline is a one-liner:

```
tty_register_ldisc (N_TCH, NULL);
```

Be aware that there are some changes in data structure referencing between the 2.4 and 2.6 kernels. For instance, the snippet that references tty->driver.write in 2.4 has to be changed to tty->driver->write if you are doing development for 2.6.

Changing a Line Discipline

The touch line discipline gets bound to the serial driver when a user space program opens the serial device connected to the touch controller. But sometimes, the user space code 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 Five opens the touch device, but changes the line discipline to N_TTY in raw mode to dump the raw data that's coming in.
LISTING FIVE: User space code to change the line discipline

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

/* At this point, our N_TCH line discipline is attached to
 * the touch controller serial port, /dev/ttySX
 */
ldisc = N_TTY;
ioctl (fd, TIOCSETD, &ldisc);

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

/*  ... */
```

The TIOCSETD ioctl() call closes the current line discipline and invokes the open entry point of the newly requested line discipline.

Further Reading

Some of the data transfer scenarios in Linux may have a data path passing through multiple line disciplines. For example, setting up a dial-up connection over Bluetooth involves the movement of data through the HCI line discipline as well as the PPP line discipline.
For a list of line disciplines supported on Linux, see include/asm/termios.h. To get a feel for other line disciplines, you might want to have a look at the corresponding line discipline source files for PPP (drivers/net/ppp_async.c), Bluetooth (drivers/bluetooth/hci_ldisc.c), IrDa (drivers/net/irda/irtty-sir.c), and SLIP (drivers/net/slip.c).

Sreekrishnan Venkateswaran has been working for IBM India since 1996. His recent Linux projects include putting Linux onto a wristwatch, a cell phone, and a pacemaker programmer. You can reach Krishnan at class="emailaddress">krishhna@gmail.com.
Fatal error: Call to undefined function aa_author_bios() in /opt/apache/dms/b2b/linux-mag.com/site/www/htdocs/wp-content/themes/linuxmag/single.php on line 62