ChatGPT

Linux 6.x TTY Driver Tutorial (Beginner to Advanced)

Overview of the TTY Subsystem

The Linux TTY subsystem is a special part of the character driver layer that handles terminal devices (serial ports, consoles, pseudoterminals, etc.). Unlike a simple character device where read/write calls go straight to the driver, TTY devices have an **extra layer called the line discipline** between user programs and the hardware driver [1]. This architecture allows the TTY core to provide common terminal functionality (like canonical input editing, echo, signal generation on Ctrl-C, etc.) independently of the low-level hardware driver [2]. In practical terms, when a process writes to a TTY, the data flows through the TTY core into a **line discipline**, then into the driver; when data comes from hardware, the driver hands it to the line discipline, which processes it and buffers it for userspace to read [1] [2]. The driver itself is mostly unaware of the line discipline – it just sends and receives raw bytes, while the line discipline implements the "policy" of how a terminal behaves.

There are a few types of TTY drivers in Linux (console drivers, serial drivers, and pseudoterminal drivers), but nearly all custom TTY drivers you'll write will be treated as serial-like devices [3]. The kernel already provides console TTYs (for virtual terminals) and PTY drivers (for pseudoterminal master-slave pairs), so new drivers typically act like serial ports. What makes TTY drivers unique is that they must integrate with the TTY core: they register themselves with the TTY subsystem, present device nodes under `/dev/` (usually as `/dev/tty*`), and cooperate with the line discipline for data handling and flow control.

Key points that distinguish TTY drivers from regular char drivers:

- **Line Discipline Layer:** This is a software module that can be "plugged" into any TTY. The default line discipline (`N_TTY`) implements typical terminal behavior: it echoes typed characters, supports line editing (canonical mode), converts special characters (e.g. `\r\n` translations), and generates signals for events like Ctrl-C [4]. Other line disciplines implement protocols such as PPP or SLIP over serial lines. All reads and writes from user space go through the line discipline [5]. The TTY driver does not directly handle cooked vs. raw input logic – that's done by the line discipline. In fact, the driver is oblivious to which line discipline is in use; it only sees a stream of bytes to send or receive [6].

- **TTY Core (tty layer):** The TTY core code mediates between user space, line disciplines, and drivers. It provides common file operations (open, read, write, ioctl, poll) for all tty devices and calls the driver's methods or the line discipline as appropriate. For example, a user `write()` goes into the TTY core, which passes the data to the line discipline's write function, which in turn calls the driver's `write` operation [1]. Similarly, when the driver receives data from hardware, it informs the core/line discipline rather than directly waking a reading process. This design means TTY drivers have to fit into a framework, rather than just implementing arbitrary read/write file operations as in a basic char driver.

- **Flow Control and Buffering:** The TTY subsystem provides standardized handling for flow control (both software XON/XOFF and hardware RTS/CTS) and buffering. The driver typically doesn't wake up reading processes or implement its own blocking logic; instead, it uses helper functions to push data to the line discipline and to check if it should stop sending data. Likewise, the subsystem maintains buffers for input and output. The driver must report how much output buffer space it has and honor requests to throttle (pause) or unthrottle (resume) data reception when buffers fill [7] [8].

Overall, writing a TTY driver means focusing on **moving raw bytes** to/from a device while the TTY core and line discipline handle the higher-level behavior. Next, we'll look at the main data structures you need to implement and how they work in Linux 6.x.

Key Data Structures: `tty_driver`, `tty_operations`, and `tty_port`

Writing a TTY driver involves interacting with a few important structures provided by the kernel:

- `struct tty_driver`**:** This is the representation of your TTY driver as a whole. It defines the range of minor numbers (i.e. how many tty devices/ports the driver supports), names for the devices, device type (serial, console, pty, etc.), and various flags and default settings. The `tty_driver` is allocated and registered with the kernel to **create the devices** (e.g. create `/dev/ttyXYZ0`, `/dev/ttyXYZ1`, etc.). It also holds pointers to the driver's operations. Think of `tty_driver` as the equivalent of the `cdev` or class in a normal char driver – it represents the driver and all its devices as a group.

- `struct tty_operations`**:** This is a table of function pointers (methods) that your driver implements to perform actual work on a port. It is analogous to file operations (`struct file_operations`) in a char driver, but tailored to TTY needs. The TTY core will call these methods in response to user actions or other events. Key operations include `open`, `close`, `write`, `write_room`, `chars_in_buffer`, `ioctl`, `set_termios`, `hangup`, and several others that we will discuss shortly. Your driver must populate a `tty_operations` structure and attach it to the `tty_driver` [9] [10].

- `struct tty_port`**:** This is a structure introduced to **represent a single TTY port (device)** and handle common port-level state and operations. Each physical or virtual serial port in your driver will have a `tty_port`. The `tty_port` layer abstracts things like the open count, carrier detection, locking, and wait queues for that port. It allows reuse of common logic across drivers. In modern kernels, **every TTY device should have an associated** `tty_port` [11]. Drivers often embed `struct tty_port` in their own device structure for each port. The `tty_port` also has its own smaller set of operations (struct tty_port_operations) for events like port activation and shutdown. Using `tty_port` helpers is highly encouraged in current kernels [12], as they handle a lot of tricky edge cases (open blocking, hangups, etc.) for you.

In summary: you will create a `tty_driver` to define your driver and devices, implement `tty_operations` for the low-level functionality, and use a `tty_port` for each device to leverage

common TTY handling code. Now, let's go step-by-step through creating and registering a TTY driver in Linux 6.x, covering all major components.

Allocating and Registering a TTY Driver (Linux 6.x)

Allocating the `tty_driver`

The first step is to allocate a `struct tty_driver` for your driver. In Linux 6.x, you do this with `tty_alloc_driver()` (replaces the older `alloc_tty_driver`). This function takes the number of devices (ports) your driver will support and a set of flags. For example, if you plan to support 4 tty devices (minor numbers 0–3 for your driver), you would call:

```
struct tty_driver *mydrv = tty_alloc_driver(4, flags);
```

Choosing the correct **flags** is important. Common flags include:

- `TTY_DRIVER_REAL_RAW` – indicates your driver will not do any special character processing (it won't, for example, translate line feed/carriage return or perform flow control in software) [13] [14]. This flag essentially promises the line discipline that your hardware is a "real" raw device and you won't unexpectedly modify data. Most serial drivers set this.

- `TTY_DRIVER_RESET_TERMIOS` – requests that when the last process closes the tty, the termios settings be reset to the driver's `init_termios` defaults [15] [16]. This is commonly used (especially for virtual/PTY drivers) to ensure that each new open gets fresh termios settings.

- `TTY_DRIVER_DYNAMIC_DEV` – **highly recommended in modern kernels** [17]. This flag means you will manually register each device (tty port) at runtime (for example, when hardware is discovered). If this flag is *not* set, the tty core will assume all possible devices always exist and will pre-create device entries for the full range of minors when you register the driver [18]. In practice, you almost always want `TTY_DRIVER_DYNAMIC_DEV` so that you only create device nodes for actual devices present. (This flag replaces the old `TTY_DRIVER_NO_DEVFS` from earlier kernels – dynamic device creation is now the default approach [19] [20].)

- Other flags: there are others like `TTY_DRIVER_UNNUMBERED_NODE` (create a single device node without a number in the name) [21], or flags specific to console/PTY behavior (`TTY_DRIVER_DEVPTS_MEM`, `TTY_DRIVER_DYNAMIC_ALLOC`, etc.), but for a basic driver you typically don't need them.

In older code (as in LDD3), one would allocate a tty driver and then manually set fields like `driver->owner` (module owner) and choose a static major number. In Linux 6.x, **you do not need to set the owner or major manually** – `tty_alloc_driver` handles the owner, and you can request a dynamic major by setting the major to 0 [20]. In fact, using static major/minor numbers is now discouraged; the kernel will allocate a major if needed. For example, the updated tiny TTY example avoids setting `owner` and uses a dynamic major instead of a fixed number [20]. This prevents conflicts and allows multiple drivers to coexist. (If you have a reserved major for your device, you can set it, but in many cases it's unnecessary.)

After allocation, you must initialize the `tty_driver` fields. Important fields include:

- `driver_name`: an internal name for your driver (often the same as your module name, e.g. `"fake_tty"`). This is used in sysfs and logging.

- `name`: the prefix for device node names. For example, if you set `name = "ttyECHO"`, your devices will appear as `/dev/ttyECHO0`, `/dev/ttyECHO1`, etc. The TTY core uses this as the base name for device creation.

- `major` and `minor_start`: the char device major number and the start of minor range for this driver. If `major` is 0, the kernel will allocate a major automatically when you register the driver. You can also use an existing major (for instance, many serial drivers use the `TTY_MAJOR` or `TTYAUX_MAJOR` for historical reasons). For simplicity, using a dynamic major (major=0) is fine – the exact number is not usually important to user space as long as the `/dev` node exists. `minor_start` is usually 0 unless you are piggy-backing off another major and need an offset.

- `type`: type of tty driver. For a serial-like driver, use `TTY_DRIVER_TYPE_SERIAL` (there are also `TTY_DRIVER_TYPE_CONSOLE` for console and `TTY_DRIVER_TYPE_PTY` for pty drivers) [22].

- `subtype`: for serial drivers, typically `SERIAL_TYPE_NORMAL` (distinguishes normal serial lines vs. things like the serial console) [22].

- `init_termios`: a `struct termios` with the initial terminal settings for new opens. You can typically start with `tty_std_termios` (a kernel global default) and then modify it if desired. For example, you might set the default baud rate (`c_cflag`) or local modes here. In older code, you'd see something like:

  ```
  driver->init_termios = tty_std_termios;
  driver->init_termios.c_cflag = B9600 | CS8 | CREAD | HUPCL | CLOCAL;
  ```

  which sets 9600 baud, 8-bit characters, enable receiver, hangup on close, and ignore modem control (CLOCAL) as default [23] [24]. You may omit this if you're fine with the kernel's default settings.

- `flags`: the flags we discussed above (REAL_RAW, RESET_TERMIOS, DYNAMIC_DEV, etc.). For example, a typical modern driver might set `flags = TTY_DRIVER_RESET_TERMIOS | TTY_DRIVER_REAL_RAW | TTY_DRIVER_DYNAMIC_DEV` [22]. (Do **not** include `TTY_DRIVER_NO_DEVFS` – that was removed; use `TTY_DRIVER_DYNAMIC_DEV` instead, as mentioned.)

- `ops`: you will assign your `tty_operations` to this (usually via `tty_set_operations()` helper). This is how the core knows about your driver's function callbacks.

Example: Suppose we are writing a simple driver for a fake serial port with 2 devices. We might do:

```
my_tty_driver = tty_alloc_driver(2, TTY_DRIVER_RESET_TERMIOS |
                                    TTY_DRIVER_REAL_RAW |
                                    TTY_DRIVER_DYNAMIC_DEV);
if (IS_ERR(my_tty_driver))
    return PTR_ERR(my_tty_driver);

my_tty_driver->driver_name = "fake_serial";
my_tty_driver->name = "ttyFAKE";
my_tty_driver->major = 0;               // dynamic major
my_tty_driver->type = TTY_DRIVER_TYPE_SERIAL;
my_tty_driver->subtype = SERIAL_TYPE_NORMAL;
my_tty_driver->init_termios = tty_std_termios;
my_tty_driver->init_termios.c_cflag = B115200 | CS8 | CREAD | HUPCL | CLOCAL;
// (115200 baud, 8-bit, enable receiver, hangup on close, no modem ctrl)
tty_set_operations(my_tty_driver, &fake_tty_ops);
```

We used `tty_alloc_driver` with flags indicating dynamic device creation and baseline raw behavior. We set a name `"ttyFAKE"` which will result in device nodes `/dev/ttyFAKE0` and `/dev/ttyFAKE1`. We left `major=0` for dynamic allocation. We also set up a default termios (115200 baud in this case). Finally, we tie in our operations vector `fake_tty_ops` (which we will define later) using `tty_set_operations()` [25].

Registering the Driver and Creating Devices

After the `tty_driver` is initialized, you must register it with the kernel:

```
retval = tty_register_driver(my_tty_driver);
if (retval) {
    tty_driver_kref_put(my_tty_driver);
    return retval;
}
```

`tty_register_driver()` tells the TTY core about your driver. If you did **not** use `TTY_DRIVER_DYNAMIC_DEV`, this call would also create device entries for all possible minors (e.g. populate `/sys/class/tty` and create device nodes for each minor). But since we passed `TTY_DRIVER_DYNAMIC_DEV` (recommended), we need to **register each device (port) individually** after registering the driver [18] [26]. In other words, with dynamic devices, `tty_register_driver` sets up the driver, and then you call another function to add each actual tty device.

Each tty device corresponds to one `struct tty_port` (one physical or logical port). So we need to set up our `tty_port` structures and then register the devices.

Setting up `tty_port`: Typically, you will have an array or dynamically allocated structure for each port. For example:

```
struct fake_serial_port {
    struct tty_port port;
    // ... plus any custom fields for your hardware, e.g. I/O memory, buffers,
etc.
} *fake_ports;
```

You allocate and initialize each port:

```
fake_ports = kzalloc(sizeof(*fake_ports) * 2, GFP_KERNEL);
for (i = 0; i < 2; ++i) {
    tty_port_init(&fake_ports[i].port);
    fake_ports[i].port.ops = &fake_port_ops;
}
```

`tty_port_init()` must be called to initialize the `tty_port` structure [27] [28]. You can also assign a `tty_port_operations` table to `port.ops` if you need to handle port-specific callbacks (we'll discuss those in the next section). If you don't have special behavior on open/close, you might not need custom port ops at all, but often you'll at least define an `activate` and `shutdown` callback.

Linking and registering devices: With the driver registered and ports initialized, the next step is to create the device nodes for each port. There are a couple of ways to do this:

- **Preferred:** Use `tty_port_register_device()` (or the variant with `_attr` if you need to add sysfs attributes). This convenience function both links the `tty_port` with the tty index and registers the device in sysfs. For each port, call:

```
struct device *dev;
dev = tty_port_register_device(&fake_ports[i].port, my_tty_driver, i, NULL);
if (IS_ERR(dev)) {
    // handle error (and undo previous registrations if needed)
}
```

This creates the `/dev/ttyFAKE{i}` device (since we set name to "ttyFAKE") and associates it with our port. Passing `NULL` for the device pointer is fine if there's no parent hardware device (e.g., for a virtual device) – otherwise you could pass a pointer to a struct device for your physical device to integrate with device model. After this call, the tty device is live and can be opened [29] [30].

- **Alternative:** In cases where `tty_port` wasn't used or available at registration time, one might use `tty_register_device()` (which takes a `struct tty_driver *`, index, and device pointer) to create the device node, and separately link the port via `tty_port_link_device()`. However, if you have a `tty_port` ready, using `tty_port_register_device` handles both linking and device creation in one call [31] [32]. (The function `tty_port_link_device()` exists for advanced scenarios, like PTYs, where ports are allocated on the fly or when you need to link without immediately registering a device node [33]. We won't need it in our simple driver.)

What registration does: When you register the driver and devices, the kernel will create entries under `/sys/class/tty/` for each tty. It will also create the device node in `/dev/` (if using devtmpfs or udev). For example, after the above calls, you might see `/dev/ttyFAKE0` and `/dev/ttyFAKE1` appear, and in `/sys/class/tty/ttyFAKE0/` you'll find a `dev` file with the major:minor numbers. The devices will also be listed in `/proc/tty/drivers` under your driver's name [34] [35].

At this point, the driver is registered and the devices exist. None of your `tty_operations` have been called yet (since nobody opened the device). The structure so far is:

- `my_tty_driver`: registered with the core.
- Two device nodes (ttyFAKE0 and ttyFAKE1) linked to `my_tty_driver`.
- Two `tty_port` structures, each associated with one device.

Before moving on, here's a condensed example of the initialization in code, putting it all together (for two devices):

```
static struct tty_driver *fake_tty_driver;
static struct fake_serial_port *fake_ports;

static int __init fake_tty_init(void)
{
    int i, ret;
    // Allocate tty_driver for 2 devices
    fake_tty_driver = tty_alloc_driver(2,
                    TTY_DRIVER_RESET_TERMIOS | TTY_DRIVER_REAL_RAW |
TTY_DRIVER_DYNAMIC_DEV);
    if (IS_ERR(fake_tty_driver))
        return PTR_ERR(fake_tty_driver);
    // Initialize tty_driver fields
    fake_tty_driver->driver_name = "fake_serial";
    fake_tty_driver->name = "ttyFAKE";
    fake_tty_driver->major = 0;
    fake_tty_driver->type = TTY_DRIVER_TYPE_SERIAL;
    fake_tty_driver->subtype = SERIAL_TYPE_NORMAL;
    fake_tty_driver->init_termios = tty_std_termios;
    fake_tty_driver->init_termios.c_cflag = B9600 | CS8 | CREAD | HUPCL |
CLOCAL;
    tty_set_operations(fake_tty_driver, &fake_tty_ops);

    // Register the tty driver
    ret = tty_register_driver(fake_tty_driver);
    if (ret) {
        tty_driver_kref_put(fake_tty_driver);
        return ret;
    }

    // Allocate and init ports
```

```
    fake_ports = kzalloc(sizeof(*fake_ports) * 2, GFP_KERNEL);
    for (i = 0; i < 2; i++) {
        tty_port_init(&fake_ports[i].port);
        fake_ports[i].port.ops = &fake_port_ops;
        // Link port with index and register /dev node
        if (!IS_ERR(tty_port_register_device(&fake_ports[i].port,
fake_tty_driver, i, NULL))) {
            continue; // success
        }
        // on error, unwind:
        while (--i >= 0) {
            tty_unregister_device(fake_tty_driver, i);
            tty_port_destroy(&fake_ports[i].port);
        }
        tty_unregister_driver(fake_tty_driver);
        tty_driver_kref_put(fake_tty_driver);
        kfree(fake_ports);
        return -EINVAL;
    }
    return 0;
}
```

Notice a few things: we used `tty_driver_kref_put()` to release the driver if registration failed (this replaced `put_tty_driver()` in newer kernels [36]). We also call `tty_port_destroy()` for each port if we unwind on error, which complements `tty_port_init()` (you either destroy ports directly or let the final `tty_port_put()` free them via the `destruct` callback, as discussed later) [37] [38].

Cleanup on Module Exit

For completeness, the module exit function should unregister the devices and driver:

```
static void __exit fake_tty_exit(void)
{
    int i;
    // Remove each device
    for (i = 0; i < 2; i++) {
        tty_unregister_device(fake_tty_driver, i);
        tty_port_destroy(&fake_ports[i].port);
    }
    // Unregister driver
    tty_unregister_driver(fake_tty_driver);
    tty_driver_kref_put(fake_tty_driver);
    kfree(fake_ports);
}
```

We call `tty_unregister_device()` for each minor we registered, which removes the device from sysfs and destroys the `/dev` node [39] [40]. Then `tty_unregister_driver()` to tell the core our driver is gone [41] [42]. Finally, we decrement the driver's reference count via `tty_driver_kref_put()` (which will free the `tty_driver` structure) and free any allocated memory. After this, any attempt to use /dev/ttyFAKE0 will result in `ENODEV` (and indeed udev/devtmpfs will remove the node).

Note: The above assumes all tty devices are closed when we unload. If any are still open, the TTY core will typically prevent rmmod from completing (or you'll get a use-count warning), since the tty_driver has active references. A robust driver might handle forced hangup of open ports on exit, but generally, it's best to ensure they're closed or not unload the module while in use.

With the driver registration in place, let's move on to implementing the actual operations that make the driver do something useful.

Implementing TTY Operations (`struct tty_operations`)

The `tty_operations` structure defines all the callbacks that the TTY core may invoke on your driver [43] [44]. We will cover the most important operations one by one, explaining what each should do in a modern driver and how it interacts with the TTY core/line discipline. For reference, here are some of the primary operations in `struct tty_operations` (from Linux 6.x):

[9]

- `open` – int `(*open)(struct tty_struct *tty, struct file *filp)`
- `close` – void `(*close)(struct tty_struct *tty, struct file *filp)`
- `write` – ssize_t `(*write)(struct tty_struct *tty, const u8 *buf, size_t count)`
- `write_room` – unsigned int `(*write_room)(struct tty_struct *tty)`
- `chars_in_buffer` – unsigned int `(*chars_in_buffer)(struct tty_struct *tty)`
- `ioctl` – int `(*ioctl)(struct tty_struct *tty, unsigned int cmd, unsigned long arg)`
- `set_termios` – void `(*set_termios)(struct tty_struct *tty, const struct ktermios *old)`
- `throttle` / `unthrottle` – void `(*throttle)(struct tty_struct *tty)`, void `(*unthrottle)(struct tty_struct *tty)`
- `stop` / `start` – void `(*stop)(struct tty_struct *tty)`, void `(*start)(struct tty_struct *tty)`
- `hangup` – void `(*hangup)(struct tty_struct *tty)`
- `flush_buffer` – void `(*flush_buffer)(struct tty_struct *tty)`
- *(plus some less commonly overridden ones like* `put_char`*,* `break_ctl`*,* `wait_until_sent`*, etc.)*

[10]

You do **not** need to implement all of these – many can be left as NULL if not applicable. We will focus on the core ones: open/close, write (and its helpers), and termios. We'll also discuss hangup and flow control (throttle/unthrottle, stop/start) conceptually.

Open and Close

Purpose: The TTY core calls your `open` when a process opens the `/dev/ttyXYZ` device file. Similarly, `close` is called when the device is closed (the last reference is gone). These are analogous to a char driver's open/release, but with some extra considerations specific to tty devices (like managing the port activation, and handling blocking opens if required).

Using `tty_port` helpers: In modern drivers, you typically do not need to write a lot of code in `open` and `close`. Instead, you can leverage the `tty_port` infrastructure:

- `tty_port_open(struct tty_port *port, struct tty_struct *tty, struct file *filp)`**:** This helper handles the common open logic. It will manage the port's open count, handle exclusive access, and even block the open if necessary (e.g., waiting for carrier detect if the tty is not *CLOCAL* and O_NONBLOCK isn't used) [45] [46]. It calls your port's `.activate` callback if this is the first open (see below), and it ensures proper locking. It returns 0 on success or -E... if open failed.

- `tty_port_close(struct tty_port *port, struct tty_struct *tty, struct file *filp)`**:** This handles the inverse operations on last close. It will call your port's `.shutdown` if this was the final close, decrement usage counts, etc.

Using these is straightforward: your `open` function finds the appropriate `tty_port` for the tty being opened, and calls `tty_port_open`, then returns its result. Your `close` function similarly calls `tty_port_close`.

Example open/close:

```
static int fake_tty_open(struct tty_struct *tty, struct file *filp)
{
    struct fake_serial_port *port = &fake_ports[tty->index];
    tty->driver_data = port;  // store pointer for later use
    return tty_port_open(&port->port, tty, filp);
}

static void fake_tty_close(struct tty_struct *tty, struct file *filp)
{
    struct fake_serial_port *port = tty->driver_data;
    tty_port_close(&port->port, tty, filp);
}
```

This is exactly the pattern used by many drivers. For instance, the `ttyprintk` pseudo-driver simply sets `driver_data` and calls `tty_port_open` in its open method [47]. The `tty->index` gives the index (minor number) of the tty, which we can use to find which port structure to use. We stash a pointer to our port structure in `tty->driver_data` for quick access in other callbacks.

Port Activation (`.activate`): When `tty_port_open()` notices that this is the first open of the port (no other process currently using it), it will call the port's `ops->activate` method if defined [12] [48]. This is your chance to "activate" the hardware: enable the device, turn on interrupts, power it up, etc. If activation fails, you can return an error and the open will be aborted. If you don't have any meaningful activation (for a virtual device, perhaps nothing needs doing), you can simply return 0. In our fake example, we might not need to do much in activate except maybe log that the port is active or reset buffer states.

Blocking opens: Traditionally, UNIX ttys can block in open if the port is not ready. For example, a real serial port might block until *Carrier Detect* is asserted (unless the application opened with O_NONBLOCK or set CLOCAL to ignore carrier). The `tty_port_open()` and related helpers handle this via `tty_port_block_til_ready()` [45] [49]. If your hardware can signal carrier (modem control), you should implement `port->ops->carrier_raised()` to return whether carrier is present, and perhaps `port->ops->dtr_rts()` to raise DTR/RTS on open. For a fake or virtual device, you can usually ignore this – if CLOCAL is set by default (as we did with CLOCAL in `c_cflag`), the open won't block. In any case, using `tty_port_open` means the kernel has taken care of these details for you.

Close and `.shutdown`: Similarly, when the last user closes the tty, `tty_port_close()` will invoke `port->ops->shutdown()` if provided [48]. This is where you should undo anything done in activate: disable interrupts, put hardware in low-power, etc. In our fake device, that might again be minimal, but a real driver might lower DTR, clear buffers, and so on. After `shutdown`, the port is considered closed; if a new open comes in, the core will call activate again.

Multiple opens: Normally, ttys are not allowed to be opened by multiple processes simultaneously *if* they are not marked `CLOCAL` and carrier is low, etc. The semantics can be tricky, but by default, `tty_port_open` ensures exclusive open unless you explicitly allow otherwise. In most cases, you can assume one active opener at a time (except for cases like PTYs where a slave and master and kernel-internal opens might coexist, but that's handled by the PTY driver logic).

Hangup vs Close: If the device is disconnected or the line discipline initiates a hangup (for example, on modem hang-up or forced hangup), the core will call your `hangup` method (discussed later). After a hangup, the file descriptor becomes unusable to user space. Your `close` will still eventually be called as the file is released, but the hangup sequence often involves cleaning up state earlier. If you use `tty_port_hangup()` in your hangup handler (as we will), it will internally call `shutdown` for you. So, typically you implement `hangup` to call `tty_port_hangup`, and you implement `shutdown` to do the actual device closing work.

Writing Data: `write`, `write_room`, `chars_in_buffer`, etc.

Purpose: The `write` operation is called when the line discipline (on behalf of a user process) wants to transmit data out through the tty. For example, when a process calls `write(fd, buf, count)` on your device, the data is first processed by the line discipline (which may transform or buffer it) and then passed to your driver via `ops->write`. In Linux 6.x, the prototype is:

```
ssize_t write(struct tty_struct *tty, const u8 *buf, size_t count);
```

It should send up to `count` bytes from the buffer `buf`. Usually, you will either copy these bytes into a hardware FIFO or transmit buffer, or if your hardware can handle it, write them out directly. The return value should be the number of bytes actually accepted/sent (often equal to `count`, unless, say, a signal interrupts or hardware buffer is full and you choose not to block – but typically the line discipline will only call `write` when there's buffer space as indicated by `write_room`).

If your driver has an internal transmit buffer or FIFO, you will often send as much as fits and return that count. The line discipline will then either queue the rest for later or try again.

Example `write`: For our fake TTY, we can implement a simple echo: whatever is written, we will immediately feed back into the input (as if the device received it). We don't have real hardware, so there is no actual transmission. Instead, we simulate that any written data arrives back on the receive line.

```
static ssize_t fake_tty_write(struct tty_struct *tty, const u8 *buf, size_t
count)
{
    struct fake_serial_port *port = tty->driver_data;
    // Simulate sending by immediately echoing to input
    for (size_t i = 0; i < count; ++i) {
        tty_insert_flip_char(&port->port, buf[i], 0);
    }
    tty_flip_buffer_push(&port->port);
    return count;
}
```

A few things to note here: - We obtained our `fake_serial_port` via `tty->driver_data` (set in open). - We looped through the provided buffer and used `tty_insert_flip_char()` to insert each byte into the **tty's flip buffer** (the input buffer managed by the tty core). The third argument is a flag (such as `TTY_NORMAL` or special flags for break or parity errors); 0 defaults to a normal character [50]. In a real driver, this insertion might happen in an interrupt handler when bytes are received, but here we are reusing it to echo output. - We then call `tty_flip_buffer_push()`, which tells the tty core that we've finished inserting pending input and it's time to push it up to the line discipline [51]. This will schedule the line discipline to consume the characters we inserted, ultimately making them available to any process reading from the tty. Essentially, we are faking an immediate loopback.

The above effectively turns writes into readable data. If you were writing a loopback test driver, this is one approach. A real serial driver, by contrast, would write bytes to the UART's transmit register or FIFO and rely on hardware interrupts to notify when more bytes can be sent or when transmission is done. In that scenario, `write` would likely just buffer the data and enable TX interrupts, `write_room` would report how much space is left in the buffer, and `chars_in_buffer` how much is still queued waiting to go out.

`write_room`**:** The TTY core (and line discipline) use the `write_room` method to ask "how many bytes can I accept for writing right now?" [52]. If `write_room` returns 0, it means the driver's output buffer is full and it cannot accept more data at the moment. The line discipline will typically stop sending data and may even block the writing process if it can't send more (or return to user space and rely on `select` / `poll` to

indicate when writable again). In a simple driver, if you have no separate hardware buffer (e.g., you can always "send" data), you can return a large number (to indicate “practically infinite” room). In our fake example, since we echo back immediately and our `write` doesn’t really queue data, we can say:

```
static unsigned int fake_tty_write_room(struct tty_struct *tty)
{
    return 65536;  // an arbitrary large value
}
```

This tells the kernel that it can send up to 64K at once without problems. For a real UART with a small FIFO, you might maintain a circular buffer for outbound data. For instance, if you have a 256-byte buffer and currently 100 bytes are in it waiting to be transmitted, `write_room` should return 156 (space left). The TTY core will only call your `write` with at most that many bytes.

`chars_in_buffer`**:** This is the converse: it should return the number of bytes still in the driver’s buffer that have not yet been transmitted [53]. The core can use this to decide if all data has been flushed (for tcdrain(), or to decide if the device is idle, etc.). In our fake driver, we don’t buffer anything in the driver (we hand everything off immediately to the flip buffer), so we can return 0 always:

```
static unsigned int fake_tty_chars_in_buffer(struct tty_struct *tty)
{
    return 0;
}
```

If you did implement a transmit buffer, this would return the number of bytes currently waiting in it.

Waking up writers (`tty_wakeup`**):** When your driver frees up space in its output buffer (e.g., hardware interrupt indicates bytes have been sent, so buffer space opens), it should call `tty_wakeup(tty)` or at least `wake_up_interruptible(&tty->write_wait)` to notify any waiting processes that they can write more [54]. The function `tty_wakeup` is a handy helper that will wake the write wait queue and also notify the line discipline (some line disciplines have a `write_wakeup` callback for additional actions) [55]. In our simple fake, since we never really block writes, we likely don't need to do this. But a real example: if `write_room` went to 0 (buffer full) and the writing process blocked, once your buffer has room again you’d do `tty_wakeup(tty)` in your transmit-complete interrupt to wake it. The `tty_core` sets up a wait queue that `select()` / `poll()` use for POLLOUT, and `tty_wakeup` will trigger those.

`put_char` **and** `flush_chars`**:** These are optional ops for handling single-character output. If `put_char` is implemented, the line discipline might use it to put a single char into your transmit buffer (for example, in canonical mode, after processing, it might output characters one by one). If you don’t implement it, the core will just buffer the char and eventually call your `write`. `flush_chars` is called to flush those buffered output chars (if `put_char` was used to enqueue them). Many simple drivers omit these and let the default mechanism handle it (the core will just call `write` with the buffered data). You can safely set these to NULL unless you have a reason to optimize single-char writes. In our case, we do not implement them (the core will route all output into our `write`).

`wait_until_sent`**:** This is called when user space invokes the TCSBRK or TIOCSERGETLSR ioctl to wait until all data is sent (e.g., `tcdrain()`). If not implemented, the core will simply wait on your chars_in_buffer becoming zero. But if your hardware can signal when it's truly empty or has a special way to wait, you might implement this. We can ignore it for our fake device.

Output flow control (`stop`/`start`): If software flow control (XON/XOFF) is enabled, or if `tcflow()` is used, the line discipline may call your `stop` method to tell you to stop sending data, and later call `start` to resume [56] [57]. Also, if the user uses Ctrl-S/Ctrl-Q on a terminal with IXON, the N_TTY line discipline will handle that by stopping output. For a basic driver, if output can be paused, you should implement these to stop/resume the transmission. For example, a UART driver might stop feeding the hardware FIFO when `stop` is called (perhaps disable the TX interrupt until `start` comes). In our fake driver, there's no real asynchronous transmitter, so `stop`/`start` would not do much. We could omit them or simply note when they're called. (Omitting is usually fine; the core will mark a flag if output is stopped. Since we echo immediately within `write`, one could argue our driver doesn't even check that flag. A more proper approach: if `stop` is called, set a flag and don't insert new flip chars in `write` until `start` clears it – but then writes would block anyway because line discipline likely wouldn't call `write` during a stop condition.)

Example: We might not implement `stop`/`start` in the fake, but a skeleton:

```
static void fake_tty_stop(struct tty_struct *tty)
{
    // e.g., set port->tx_stopped = true;
}

static void fake_tty_start(struct tty_struct *tty)
{
    // e.g., set port->tx_stopped = false;
    // possibly kick the transmitter if there is buffered data
}
```

If you handle XON/XOFF in hardware or differently, you may also need to handle the `send_xchar` operation (which is invoked to send an XON or XOFF char immediately in some cases) – we will not delve into that here.

Reading Data: Handling Input from the Device

One peculiarity of TTY drivers is that **there is no `read` method in `tty_operations`**. User read requests are handled by the line discipline, which reads from its input buffers. So how does data get into those buffers? The driver is responsible for capturing input from hardware (e.g., from interrupts or polling) and pushing it into the TTY core's buffers. This is done with the functions we just used in our echo example: `tty_insert_flip_char()` (or related helpers) and `tty_flip_buffer_push()`.

Typical input path (with real hardware): Suppose you have a UART that receives bytes via an interrupt. In the interrupt handler, you would fetch the byte from a register, and then call something like:

```
tty_insert_flip_char(&my_port->port, received_byte, 0);
```

You might do this in a loop for all bytes received. There are also functions like `tty_insert_flip_string` to copy a whole buffer of bytes at once for efficiency. If the byte had a framing or parity error, you can pass flags like `TTY_FRAME` or `TTY_PARITY` instead of 0, which the line discipline (N_TTY) can use to decide how to handle (often it generates a special character like `\0` for parity errors if `INPCK` is set, etc.). After inserting all available bytes into the flip buffer, you call:

```
tty_flip_buffer_push(&my_port->port);
```

This signals the core/ldisc that "input is available now." The N_TTY line discipline will then take those characters out of the flip buffer, process them (do canonical editing, echo them back out if echo is enabled, etc.), and queue them for user space to read. If a process is blocked in read or waiting via poll, the arrival of data will wake it up.

For our fake driver, we did exactly this but in the `write` function (to echo outgoing data). In a real driver, you'd do it in the receive interrupt or a scheduled work. The line discipline runs in process context (not in the interrupt), so pushing the buffer schedules a softirq or work to handle the data safely.

Flow control – `throttle` **/** `unthrottle` **:** If the line discipline's input buffer fills up (for instance, a process isn't reading fast enough and the buffer hits a high watermark), the core may call your `throttle` method [10]. This is a signal for the driver to **stop reading data from the hardware** if possible. For a UART, you might disable the RX interrupt or hardware flow control (e.g., raise RTS line to tell the sender to pause, or send an XOFF character if using software flow control). When the user reads some data and the buffers go below the low watermark, the core calls `unthrottle` so you can resume receiving (re-enable interrupts or drop RTS, or send XON). It's important to implement these if your hardware supports flow control or if dropping data is unacceptable. If not implemented, the system might drop characters or just rely on higher-level flow (which could mean data loss). For a fake device that generates data internally, you can likely ignore throttle/unthrottle or just log it. But for completeness:

```c
static void fake_tty_throttle(struct tty_struct *tty)
{
    // e.g., stop generating or buffering new input data
}

static void fake_tty_unthrottle(struct tty_struct *tty)
{
    // resume input generation
}
```

In a loopback or echo driver, you might not generate input except in response to writes, so throttle/unthrottle might not be meaningful. However, imagine a driver that generates a continuous stream of data (like a GPS receiver). When throttle is called, you might pause polling or buffering to avoid overflow.

Buffer sizes: The flip buffer (also called the “tty buffer”) has a finite size (commonly 2*64 bytes as a double buffer by default in N_TTY, historically 4KB for canonical buffer). The throttle mechanism helps manage not overrunning these.

In-band signals: If your driver detects a special condition (like a break signal on the line or a parity error), you can insert appropriate flags or special bytes. For example, a break condition might be reported by calling `tty_insert_flip_char(port, 0, TTY_BREAK)`. The N_TTY line discipline will translate that into a `\0` (ASCII NULL) if `IGNBRK` is not set and will raise a SIGINT or SIGBREAK to processes if `BRKINT` is enabled, etc. The driver itself doesn’t handle the signal; it just reports the break condition and the line discipline and core handle the rest [58].

Termios (speed/format) changes affecting input: If the user changes termios settings (e.g., disabling parity checking or changing character size), your driver should respect that for subsequent input. We handle that in `set_termios` (discussed next).

`ioctl` and `set_termios`

`ioctl`**:** The TTY layer handles many ioctls itself (the common ones in `<linux/tty.h>` and `<linux/serial.h>`), but any that are not recognized or are hardware-specific will be passed to your driver’s `ioctl` callback [59]. For example, TTY ioctls like TIOCMGET (get modem status lines) and TIOCMSET (set DTR/RTS lines) can be implemented by the driver or by using the provided `tiocmget`/`tiocmset` hooks in tty_operations [60] [61]. If you implement those specific hooks, the core will use them and not call your generic `ioctl` for those. In other cases, you might receive ioctl calls for setting custom baud rates, controlling loopback modes, etc.

In our fake driver, we might not have any custom ioctls, so we could either not implement `ioctl` at all (returning -ENOIOCTLCMD by default), or implement it to handle a few common ones if we want to simulate them. For instance, if we want to support reading modem control state (even if fake), we could implement `tiocmget` to return, say, TIOCM_CAR (carrier) always on if we treat the device as always connected.

For a beginner tutorial, we can simply note: if the driver doesn’t need special control commands, you can leave `ioctl` NULL and the core will handle standard ones or return ENOTTY to the app for unhandled cases. If you do implement it, ensure to handle commands carefully and use copy_{from,to}_user if needed for data.

`set_termios`**:** This method is called when the user changes terminal settings via `tcsetattr()` (termios changes) [62] [63]. It provides the new settings already in `tty->termios`, and a pointer to the old termios so you can see what changed. The typical use of `set_termios` in a serial driver is to check for changes in baud rate, parity, number of stop bits, etc., and then reconfigure the hardware accordingly. For example, if the baud rate changed, you’d call your hardware-specific routine to set the new baud divisor, or if parity changed from none to even, you’d update the UART control register.

In our fake driver, there is no real hardware to configure. We can still implement `set_termios` to illustrate what one *would* do:

```
static void fake_tty_set_termios(struct tty_struct *tty, const struct ktermios
*old_termios)
{
    unsigned int old_cflag = old_termios->c_cflag;
    unsigned int new_cflag = tty->termios.c_cflag;
    // Check what changed
    if ((old_cflag ^ new_cflag) & CSIZE) {
        // character size changed (e.g., 7-bit vs 8-bit)
    }
    if ((old_cflag ^ new_cflag) & (PARENB | PARODD)) {
        // parity setting changed
    }
    if (old_termios->c_ospeed != tty->termios.c_ospeed) {
        // output baud rate changed
        unsigned int baud = tty_get_baud_rate(tty);
        printk(KERN_INFO "fake_tty: Baud rate changed to %u\n", baud);
    }
    // In a real driver, configure hardware registers here.
}
```

The above checks flags in `c_cflag` (control flags) for data bits (`CS5`–`CS8`), parity enable (`PARENB`), parity type (`PARODD`), etc. It also checks the baud rate. The kernel provides helper functions like `tty_get_baud_rate(tty)` which returns the new baud (it looks at `c_ospeed` or computed from termios structure) – this is easier than decoding bits manually. If your hardware cannot support the requested baud exactly, you should set `tty_termios_encode_baud_rate(tty, new, new)` with the actual rates you configured, so user space knows the effective rate. Also, if the input baud (`c_ispeed`) is 0, it generally means "use the same as output baud."

Since our device is virtual, we don't actually *change* anything, but we might log it. In practice, one might ignore termios changes they can't support (but still update the `tty->termios` to reflect them). The TTY core will take care of things like ICANON, IXOFF at the software level; the driver mainly cares about hardware-related flags in c_cflag (baud, data bits, stop bits, parity, flow control bits like CRTSCTS).

Termios structure fields of interest to driver: `c_cflag` (baud rate bits, data bits (CS5-8), stop bits (CSTOPB), parity (PARENB/PARODD), hardware flow (CRTSCTS), CLOCAL (ignore modem control), CREAD (enable receiver)), and maybe `c_iflag` if you want to know about IGNBRK (to decide whether to ignore break conditions) or IXON (maybe to let hardware handle XON/XOFF if possible). However, these input flags are largely handled in the line discipline, so driver mostly looks at c_cflag and sometimes `c_iflag & INPCK` to decide whether to do parity checks.

If your driver has limitations (e.g., only supports one baud rate or does not support parity), you should document or possibly adjust the termios to what you *can* do. For example, if parity is requested but you can't do it, you may want to clear PARENB in `tty->termios`. Usually, you inform the user by leaving the unsupported settings unchanged after the call, or by adjusting to nearest supported.

For our tutorial, we implement a minimal `set_termios` just to show hooking it up (maybe printing debug info).

Hangup

Purpose: Hangup is called when the tty is to be disconnected **as if the phone hung up** – in modern terms, this often means a USB serial device was unplugged, or the kernel wants to invalidate the line (for example, controlling tty of a process session is being closed). Hangup can be initiated by setting `TTY_IOC_HANGUP` ioctl or by the kernel (for instance, when a PTY master closes, the slave gets hung up).

When a hangup happens, the TTY core will call your `ops->hangup`. The driver should perform an immediate shutdown of the port and any associated hardware, similar to a final close. It should also *signal to the tty core that the hangup is done.* The easiest way to do this is to use the helper:

```
tty_port_hangup(&my_port->port);
```

This function will mark the port as hung up, drop the line discipline, and wake up any blocked readers/writers with an error (so they get EHangup). It also invokes the low-level `shutdown` (via the port ops) for you [64] [65]. Essentially, it does the heavy lifting of a hangup. In most drivers, therefore, the `hangup` method simply looks up the tty's port and calls `tty_port_hangup()` [66].

Example:

```
static void fake_tty_hangup(struct tty_struct *tty)
{
    struct fake_serial_port *port = tty->driver_data;
    tty_port_hangup(&port->port);
}
```

This will ensure that any future operations on that file descriptor return `EIO` and that the line is reset. The next open will get a fresh line discipline.

Internally, `tty_port_hangup` will call our `fake_port_ops.shutdown` (if defined) to shut the device, similar to a close. It also manages reference counts to potentially free the port structure if it was dynamically allocated and no longer used.

When hangup is used: For hardware serial, if the device loses DCD (carrier detect) and `HUPCL` is set, the line discipline might initiate a hangup. For USB serial, if the device is unplugged, the USB driver likely calls `tty_port_hangup` to terminate any ongoing I/O. For PTYs, when the master is closed, the slave is hung up.

Since our fake device is virtual and not truly disconnectable (unless we simulate it), hangup will likely only occur if explicitly invoked or on module unload (the tty core might hangup on unregister). It's still good practice to implement it as above.

`struct tty_port_operations`: Activate, Shutdown, Destruct, etc.

We mentioned the port ops earlier. To tie everything together:

- `activate(struct tty_port *port, struct tty_struct *tty)`: Called on first open (not every open). Return 0 for success or an error. Here you can allocate resources needed only when the port is in use (e.g., start a kthread or allocate a device buffer, enable hardware). If you fail here, open will fail. In our fake, we might not need to allocate anything, so we just return 0. If we wanted to simulate something, we could start a periodic timer here to generate input data (just as an idea for a more complex tutorial exercise).

- `shutdown(struct tty_port *port, struct tty_struct *tty)`: Called on last close or on hangup. This should disable the port and free resources allocated at activate. For example, stop any timers, disable interrupts, free DMA, etc. After this, the device should be quiescent.

- `destruct(struct tty_port *port)`: Called when the tty_port is being freed (final put). If you dynamically allocated your port structure (or other memory), this is where to free it. If your `fake_serial_port` is `kmalloc`ed per device, you could set `destruct` to a function that does `kfree(container_of(port, struct fake_serial_port, port))`. In our example, we allocated an array and free it in module exit, so we might not use destruct.

- `carrier_raised` **and** `dtr_rts`: These are optional but used for open blocking logic. `carrier_raised(port)` should return whether the physical carrier is present (for modems). `dtr_rts(port, on)` should assert or drop the DTR (and possibly RTS) lines for modem control. If our device doesn't support modem control, we can leave these NULL. If we left CLOCAL in c_cflag by default (which we did, by setting CLOCAL), the absence of these is fine. If not, and if carrier_raised is NULL, the core assumes carrier is always on.

- `notify_framing` **/** `notify_break` **(in newer kernels)**: there are hooks to notify the port of a framing error or break – but if you use `tty_insert_flip_char` with flags, you may not need these.

For our fake driver, we can implement a minimal port ops:

```
static int fake_port_activate(struct tty_port *port, struct tty_struct *tty)
{
    // e.g., initialize hardware or buffers
    return 0;
}
static void fake_port_shutdown(struct tty_port *port, struct tty_struct *tty)
{
    // e.g., disable hardware or timers
}
static const struct tty_port_operations fake_port_ops = {
    .activate  = fake_port_activate,
    .shutdown  = fake_port_shutdown,
```

```
    // .destruct not needed since we free manually
};
```

We assigned this `fake_port_ops` earlier to each port (`fake_ports[i].port.ops = &fake_port_ops`). The TTY core will call these at the appropriate times (via `tty_port_open` / `close` / `hangup` as discussed).

Pseudoterminal (PTY) Handling and Terminal Emulators

Before concluding, a quick overview of how pseudoterminals differ and integrate, since it was listed as a topic:

A **pseudoterminal (PTY)** is a pair of virtual devices that provide a bidirectional data channel, used typically by terminal emulator programs. One end is the **master** (often accessed via `/dev/ptmx` which gives you a PTY master), and the other end is the **slave** (which appears as a `/dev/pts/<n>` device and acts like a regular tty device from the perspective of applications). The PTY slave end is a TTY driven by the kernel's `drivers/tty/pty.c` driver, and it uses the standard line discipline (so that programs like shells run on it see a normal terminal). The master end is usually handled by a user-space program (like `xterm`, `screen`, or SSH server), which reads from and writes to it. Data written to the PTY master appears as input on the slave, and data written to the slave appears for reading on the master [67] [68]. Essentially, the PTY driver connects two tty_structs back-to-back.

The Linux PTY driver is already implemented, so you typically don't write one from scratch. But if you were to integrate a PTY-like function in your driver (say, to create a bridge between your custom hardware and a terminal emulator), you might use similar techniques: on one side, act as a tty device that programs can open; on the other side, perhaps forward data to another interface.

Our tutorial driver is not a PTY, but understanding that PTYs use the same TTY infrastructure helps clarify why certain things (like line disciplines, resets on hangup, etc.) exist. For example, when an `xterm` closes, the kernel generates a hangup on the PTY slave, which causes the shell on that slave to get SIGHUP.

In a sense, the PTY slave is another kind of TTY driver (with its own `tty_driver` called "pty_slave") and the PTY master is a special character device that feeds into it. They are paired internally by the kernel. If you look at `/proc/tty/drivers`, you'll see entries for pty_master and pty_slave with their device ranges [69].

Integration with terminal emulators: If you wanted your fake device to interact with a terminal emulator, one approach could be to have your driver create a PTY master internally and shuttle data between it and your fake tty. But that's beyond the scope here. Generally, terminal emulators will just open `/dev/ptmx` to get a PTY. For testing our driver, we can simply use common tools like `cat` or `echo` in a shell.

Minimal Example Code Recap

Bringing it all together, here is a summary of a minimal TTY driver (for a single port, for brevity) updated to Linux 6.x APIs:

```c
#include <linux/module.h>
#include <linux/tty.h>
#include <linux/tty_driver.h>
#include <linux/tty_port.h>

struct fake_serial_port {
    struct tty_port port;
};

static struct tty_driver *fake_tty_driver;
static struct fake_serial_port fake_port;  // single port example

/* Port ops */
static int fake_port_activate(struct tty_port *port, struct tty_struct *tty)
{
    // Initialize hardware (if any). For fake, nothing.
    return 0;
}
static void fake_port_shutdown(struct tty_port *port, struct tty_struct *tty)
{
    // Disable hardware or free resources. For fake, nothing.
}
static const struct tty_port_operations fake_port_ops = {
    .activate = fake_port_activate,
    .shutdown = fake_port_shutdown,
};

/* TTY ops */
static int fake_tty_open(struct tty_struct *tty, struct file *file)
{
    tty->driver_data = &fake_port;
    return tty_port_open(&fake_port.port, tty, file);
}
static void fake_tty_close(struct tty_struct *tty, struct file *file)
{
    tty_port_close(&fake_port.port, tty, file);
}
static ssize_t fake_tty_write(struct tty_struct *tty, const u8 *buf, size_t
count)
{
    // Echo back input
    for (size_t i = 0; i < count; ++i) {
        tty_insert_flip_char(&fake_port.port, buf[i], 0);
    }
    tty_flip_buffer_push(&fake_port.port);
    return count;
}
```

```
static unsigned int fake_tty_write_room(struct tty_struct *tty)
{
    return 4096;  // arbitrary space available
}
static unsigned int fake_tty_chars_in_buffer(struct tty_struct *tty)
{
    return 0;  // no output buffer
}
static void fake_tty_set_termios(struct tty_struct *tty, const struct ktermios
*old)
{
    // Here we could apply baud/parity changes. We'll just print new baud for
demo.
    pr_info("fake_tty: new baud rate %d\n", tty_get_baud_rate(tty));
}
static void fake_tty_hangup(struct tty_struct *tty)
{
    tty_port_hangup(&fake_port.port);
}

static const struct tty_operations fake_tty_ops = {
    .open = fake_tty_open,
    .close = fake_tty_close,
    .write = fake_tty_write,
    .write_room = fake_tty_write_room,
    .chars_in_buffer = fake_tty_chars_in_buffer,
    .set_termios = fake_tty_set_termios,
    .hangup = fake_tty_hangup,
    // .ioctl = ... (optional)
};

static int __init fake_tty_init(void)
{
    int ret;
    fake_tty_driver = tty_alloc_driver(1, TTY_DRIVER_RESET_TERMIOS |
                                          TTY_DRIVER_REAL_RAW |
                                          TTY_DRIVER_DYNAMIC_DEV);
    if (IS_ERR(fake_tty_driver))
        return PTR_ERR(fake_tty_driver);

    fake_tty_driver->driver_name = "fake_tty";
    fake_tty_driver->name = "ttyFAKE";
    fake_tty_driver->major = 0;  // dynamic major
    fake_tty_driver->type = TTY_DRIVER_TYPE_SERIAL;
    fake_tty_driver->subtype = SERIAL_TYPE_NORMAL;
    fake_tty_driver->init_termios = tty_std_termios;
    fake_tty_driver->init_termios.c_cflag = B9600 | CS8 | CREAD | HUPCL |
CLOCAL;
```

```
    tty_set_operations(fake_tty_driver, &fake_tty_ops);

    tty_port_init(&fake_port.port);
    fake_port.port.ops = &fake_port_ops;

    ret = tty_register_driver(fake_tty_driver);
    if (ret) {
        tty_driver_kref_put(fake_tty_driver);
        return ret;
    }
    // Register the single device (index 0) for this driver
    if (IS_ERR(tty_port_register_device(&fake_port.port, fake_tty_driver, 0,
NULL))) {
        tty_unregister_driver(fake_tty_driver);
        tty_driver_kref_put(fake_tty_driver);
        return -ENOMEM;
    }
    pr_info("fake_tty: driver initialized\n");
    return 0;
}

static void __exit fake_tty_exit(void)
{
    tty_unregister_device(fake_tty_driver, 0);
    tty_unregister_driver(fake_tty_driver);
    tty_driver_kref_put(fake_tty_driver);
    tty_port_destroy(&fake_port.port);
    pr_info("fake_tty: driver removed\n");
}

module_init(fake_tty_init);
module_exit(fake_tty_exit);
MODULE_LICENSE("GPL");
```

This code creates a `/dev/ttyFAKE0` device. Opening it will succeed and not block (because we set CLOCAL to ignore carrier). Writing to it will echo data back (so you can do `cat /dev/ttyFAKE0` in one terminal and then echo something into `/dev/ttyFAKE0` from another, and the first will see it). It handles basic termios changes and cleans up on exit. We used the recommended new APIs: `tty_alloc_driver`, `tty_driver_kref_put`, and the `tty_port_*` helpers instead of deprecated calls. No `owner` field is set (tty_alloc_driver did that internally) [20]. We chose dynamic major and dynamic device registration for best practice.

Tutorial Exercise: Building a Fake TTY Device

As a final exercise, you can try to expand this minimal driver into a fully working module and test it on a Linux 6.x system (e.g. on QEMU or real hardware):

- **Goal:** Implement a **fake TTY driver** that creates one or more `/dev/ttyYOURNAME*` devices. When data is written to these devices, the driver should simulate some behavior – for example, echoing the data back (like a loopback), or perhaps converting it to uppercase before echoing, etc. The device should be readable and writable: a process reading from it should get whatever data was written (perhaps transformed).

- **Device visibility:** After loading your module, verify that the device node appears (e.g., `/dev/ttyFAKE0`). This is handled by `tty_register_driver` and `tty_port_register_device` creating the sysfs and dev entries. Ensure you have `CONFIG_DEVTMPFS` or udev running so that the device file is created automatically.

- **Basic functionality test:** You can use two shells. In one, run `cat /dev/ttyFAKE0`. It will block waiting for input. In another, run `echo "Hello" > /dev/ttyFAKE0`. The first shell running `cat` should output "Hello". This would mean your write->echo->read path is working. Try also writing large amounts of data or using `dd` to see that `write_room` is respected (the writing process shouldn't hang indefinitely unless you deliberately throttle).

- **Callbacks to implement:** At minimum, you need `open`, `close`, `write`, `write_room`, `chars_in_buffer`, and maybe `hangup`. We also suggest implementing `set_termios` (even if just to acknowledge changes) and possibly `ioctl` if you want to handle any particular commands. The template above covers these.

- **Simulating functionality:** If echoing is too trivial, you could simulate a simple **serial device** that, say, responds with a fixed message. For example, if data is written, the driver could ignore the content and always return `"OK\r\n"` or some canned response after a delay. This might involve using `schedule_work` or a timer to push data after a short timeout (to simulate asynchronous response). Make sure to use `tty_insert_flip_char` in the work function to inject the response.

- **Flow control:** For an advanced challenge, implement throttle/unthrottle to see it in action. You could keep an internal counter of bytes received but not yet read by user. If it exceeds some threshold, call `tty_insert_flip_char(port, 0, TTY_STOP)` to send an XOFF to the sender (this is how N_TTY signals software flow control), or simply stop echoing until `unthrottle` is called. This is tricky to test; using `dd` or a custom program to write a lot and seeing if the reading side slows down might be necessary.

- **Multi-port support:** Modify the driver to create, say, 4 devices (`ttyFAKE0`–`ttyFAKE3`). This means using `tty_alloc_driver(4, ...)`, and looping to register each with `tty_port_register_device`. Test that you can use each independently. Each `tty_struct->index` will correspond to a different fake_port structure in an array.

- **Integration with console (optional):** For fun, you could register your device as a console (the ttyprintk driver did this). This involves `register_console` and providing a `console.write` callback. That way you could use `console=ttyFAKE0` at boot. This is advanced and optional.

By completing this, you will have a deeper understanding of the TTY subsystem. You saw how the **tty core** and **line discipline** handle the heavy lifting (buffering, canonical mode, signals) and the driver mainly pushes bytes in and out. You used modern APIs like `tty_alloc_driver` and `tty_port` helpers which simplify driver code and handle edge cases. This basic framework can be extended to real hardware drivers – for example, replacing the echo logic with actual UART register reads/writes, and using interrupts to trigger `tty_insert_flip_char` calls.

Testing notes: QEMU is a good environment – you can load your module in a VM and interact with the tty device. If using QEMU, since this is not a physical UART, it won't appear unless your module creates it (which it does via registration). Just ensure your kernel has device filesystem support so the node appears. Use `dmesg` to see the debug prints (like from `set_termios`). If something isn't working (e.g., you write but nothing is read), add some `printk` in your ops to trace the flow.

Finally, inspect `/proc/tty/drivers` after loading your module – you should see an entry for your driver, showing the name, the device nodes, major number, and type. For example, it might list `fake_tty /dev/ttyFAKE 240 0-0 serial` (if it got major 240 and one minor) [70] [71]. This helps confirm registration succeeded.

Good luck with your implementation, and happy hacking on the TTY subsystem! With this knowledge, you can create virtual serial devices, debugging consoles, or interface real serial hardware to Linux. The TTY core may be old, but it's a powerful abstraction that remains important in modern kernels. [72] [73]

---

[1] [2] [3] [6] [7] [8] [23] [24] [34] [35] [50] [51] [54] [58] [69] [70] [71] ch18.pdf
file://file_000000002370722fb29cc827d2d2c0dd

[4] [5] jlam | Understanding the tty subsystem: Line discipline
https://lambdalambda.ninja/blog/56/

[9] [10] [11] [13] [14] [15] [16] [17] [18] [21] [25] [26] [29] [30] [31] [32] [33] [39] [40] [41] [42] [43] [44] [52] [53] [56] [57] [59] [60] [61] [62] [63] TTY Driver and TTY Operations — The Linux Kernel documentation
https://docs.kernel.org/driver-api/tty/tty_driver.html

[12] [27] [28] [37] [38] [45] [46] [48] [49] [64] [65] TTY Port — The Linux Kernel documentation
https://docs.kernel.org/driver-api/tty/tty_port.html

[19] [20] [36] [72] [73] Working on LDD3's tiny tty example – bowfinger.de
https://bowfinger.de/blog/2024/04/working-on-ldd3s-tiny-tty-example/

[22] drivers/tty/goldfish.c - kernel/common - Git at Google
https://android.googlesource.com/kernel/common/+/upstream-f2fs-stable-linux-4.4.y/drivers/tty/goldfish.c

[47] [66] drivers/char/ttyprintk.c - kernel/common - Git at Google
https://android.googlesource.com/kernel/common/+/refs/tags/android16-6.12-2025-06_r19/drivers/char/ttyprintk.c

55 TTY Struct - The Linux Kernel documentation
https://docs.kernel.org/driver-api/tty/tty_struct.html

67 68 jlam | Understanding the tty subsystem: Overview and architecture
https://lambdalambda.ninja/blog/54/