

SNULL

Wednesday, October 29, 2025 9:46 AM

Excellent question — **snnull is not a hardware driver, but a sample (dummy) network driver** written for **learning purposes**, especially for people studying Linux network driver development. It's one of the **most important examples** in the famous book:

▀ ***Linux Device Drivers, 3rd Edition (LDD3)*** by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman.

Let's break it down in deep detail ↗

❖ What is snnull?

snnull = Simple Network Utility (or sometimes “simple null”)

It is a **toy (dummy) network driver** that simulates two Ethernet interfaces (e.g. snull0 and snull1), which "talk" to each other entirely in memory — no physical network card involved.

It's designed purely for **educational purposes** to demonstrate:

- How to register a network interface in the Linux kernel
- How to send and receive packets
- How to integrate with the kernel's network stack
- How interrupts and NAPI work in a real driver

So:

⌚ **snnull helps you learn the full flow of a real network driver without needing hardware.**

⚙️ What snnull Does Internally

When you load the snnull module, it creates **two virtual Ethernet devices**:

snull0

snull1

Now, if you send data (e.g., ping) to one of them, the packet never leaves the computer — instead, snnull internally copies the packet to the **other interface**, pretending it was “received from the network”.

For example:

ping snull1

- sends a packet through snull0
- snull0 “transmits” the packet (software copy)
- snull1 “receives” that same packet

- kernel delivers it up the network stack
- So it simulates a **two-device loopback connection**, similar to a local network.

Core Components of snull

Component	Description
struct net_device	Represents each fake Ethernet device (snull0, snull1)
struct net_device_ops	Contains callback functions for open, close, transmit, etc.
snnull_open()	Starts the device, enables queues
snnull_stop()	Stops the device, disables queues
snnull_tx()	Called when upper layer wants to transmit
snnull_rx()	Simulates packet reception by copying data to the peer device
snnull_interrupt()	Simulates an interrupt from “hardware” when RX complete
snnull_setup()	Initializes MAC address, MTU, etc.
snnull_cleanup()	Removes the devices and frees memory

Flow of Operation (Simplified)

1. Driver Initialization

- Allocates two net_device structures using alloc_etherdev()
- Registers them using register_netdev()
- Sets up fake MAC addresses (like 00:0a:11:22:33:44)

2. Open Device (ifconfig snull0 up)

- Calls snnull_open()
- Starts transmission queue with netif_start_queue()

3. Send Packet

- Kernel calls snnull_tx()
- It copies the packet into an internal buffer
- Then “delivers” it to the peer device (snnull1) using a simulated interrupt

4. Receive Packet

- snnull_rx() allocates an skb
- Fills it with the copied data
- Calls netif_rx() to push it into the kernel stack

5. Cleanup

- On rmmod, unregisters both devices and frees memory

Why Use snull?

Purpose	Benefit
Learning	Perfect for studying how a network driver interacts with the kernel

Debugging	Lets you practice with net_device, skb, and interrupts safely
Experimentation	Can modify to test NAPI, DMA, or ethtool functions
Simulation	Acts like a real Ethernet card (shows up in ifconfig, ip link)

Key Files (in LDD3 version)

If you check the original LDD3 code, you'll find:

snnull.c – main driver source
 Makefile – build script
 load/unload – helper shell scripts

You can download it (if you're reading the book) or find a modern version on GitHub adapted for recent kernels.

What You Learn From It

By studying snnull, you learn:

- How the Linux kernel represents network devices (net_device)
- How transmission (ndo_start_xmit) and reception (netif_rx) work
- How to use socket buffers (struct sk_buff)
- How to use spinlocks, atomic variables, and interrupt contexts
- How to test a driver without hardware

Example Usage

After building and inserting snnull.ko:

```
sudo insmod snnull.ko
ifconfig snull0 192.168.0.1 up
ifconfig snull1 192.168.0.2 up
ping 192.168.0.2 -I snull0
```

You'll see packets being "sent" and "received" — all inside your RAM.

Summary

Concept	Meaning
Name	snnull (Simple Network Utility)
Type	Virtual / dummy network driver
Purpose	Educational (example for learning Linux network stack)

Implements	net_device, TX/RX callbacks, interrupts, etc.
Creates	Two fake Ethernet devices (snull0, snull1)
Behavior	Simulates packet transmission between them
Found in	Linux Device Drivers, 3rd Edition (LDD3)
Real hardware?	✗ No (completely virtual)

Excellent — you’re diving into the **heart of how network drivers (like snull) register themselves with the Linux kernel**. Let’s break this section down *completely* — explaining every step, function, and struct involved, using your reference (from *Linux Device Drivers, 3rd Edition*, Chapter 17) but in a **modern, clear, step-by-step manner**.

❖ Goal of This Section

To understand **how snull registers itself as a network device** in the Linux kernel — that is, how it goes from being a kernel module to showing up in ifconfig or ip link as an interface like sn0 or sn1.

- ◊ [1. Background: What Happens When You Load a Network Driver](#)

When you insmod snull.ko, the following happens:

6. The module’s **initialization function** runs (e.g. snull_init_module()).
7. The driver:
 - Allocates a struct net_device for each interface it supports.
 - Initializes each device.
 - Registers them with the kernel networking subsystem.
8. The kernel now knows about your new interfaces (sn0, sn1, etc.) and userspace tools can see them.

Unlike **char** or **block drivers**, network drivers:

- **Do not use major/minor numbers.**
- **Are identified by interface names** like eth0, sn0, etc.
- Are represented in the kernel by **struct net_device**.

- ◊ [2. Core Structure: struct net_device](#)

Defined in <linux/netdevice.h>.

It represents one **network interface**.

Common Important Fields

Field	Description
char name[IFNAMSIZ]	Interface name (like "sn0")
unsigned long state	Device state (up/down, etc.)
struct net_device_ops *netdev_ops	Pointer to function table (open, stop, start_xmit, etc.)
struct net_device_stats stats	Tx/Rx counters

unsigned char dev_addr[ETH_ALEN]	MAC address
void *priv	Private data for your driver (cast to your own struct)

◊ 3. Private Data (Driver-Specific State)

Each network driver usually needs to keep its own per-device data — e.g., TX/RX buffers, spinlocks, stats, etc.

snull defines:

```
struct snull_priv {
    struct net_device_stats stats;
    int status;
    struct sk_buff *skb;
    spinlock_t lock;
    /* ... more fields ... */
};
```

You can attach this to the network device via alloc_netdev() (or alloc_etherdev()).

◊ 4. Allocation — alloc_netdev()

```
struct net_device *alloc_netdev(
    int sizeof_priv,
    const char *name,
    void (*setup)(struct net_device *)
);
```

Parameters:

Parameter	Meaning
sizeof_priv	Size of your driver's private data (struct snull_priv)
name	Interface name (can include %d for numbering)
setup	Initialization function (sets default values in net_device)

When you call alloc_netdev(), the kernel:

- Allocates memory for both struct net_device **and** your private structure.
- Zeros them.
- Calls your setup() function to configure defaults.

In snull:

```
snull_devs[0] = alloc_netdev(sizeof(struct snull_priv), "sn%d", snull_init);
snull_devs[1] = alloc_netdev(sizeof(struct snull_priv), "sn%d", snull_init);
```

This allocates two virtual network devices, sn0 and sn1.

◊ 5. Initialization — `snull_init()`

This is your **setup function**, passed to `alloc_netdev()`.

```
void snull_init(struct net_device *dev)
{
    /* Assign network operations */
    dev->netdev_ops = &snull_netdev_ops;

    /* Assign hardware (MAC) address */
    dev->dev_addr[0] = 0x00;
    dev->dev_addr[1] = 0x0C;
    dev->dev_addr[2] = 0x29;
    dev->dev_addr[3] = 0x3E;
    dev->dev_addr[4] = 0x5B;
    dev->dev_addr[5] = 0x11;

    /* Initialize private data */
    struct snull_priv *priv = netdev_priv(dev);
    memset(priv, 0, sizeof(struct snull_priv));
    spin_lock_init(&priv->lock);

    /* Other optional setups */
    ether_setup(dev); // sets Ethernet-specific fields (MTU, type, flags)
}
```

Note: If you use `alloc_etherdev()` instead of `alloc_netdev()`, it automatically calls `ether_setup()` for you.

◊ 6. Registering the Device — `register_netdev()`

Once initialization is done, you must register it with the kernel:

```
for (i = 0; i < 2; i++) {
    int result = register_netdev(snull_devs[i]);
```

```

if (result)
    printk("snnull: error %i registering device \'%s\'\n",
        result, snull_devs[i]->name);
}

```

After this call:

- The kernel adds your device to its global list of network devices.
- Userspace can now see it via:
`ip link show`

or

`ifconfig sn0 up`

⚠ Important:

Once `register_netdev()` succeeds, your device can immediately receive callbacks — e.g.:

- `ndo_open()` if user runs `ifconfig sn0 up`
- `ndo_start_xmit()` when packets are sent

So you must ensure all initialization is **complete before** registration.

◊ [7. Network Device Operations — struct net_device_ops](#)

Defined in `<linux/netdevice.h>`.

It's like the `file_operations` structure for char drivers.

Example from snull:

```

static const struct net_device_ops snnull_netdev_ops = {
    .ndo_open      = snnull_open,
    .ndo_stop     = snnull_release,
    .ndo_start_xmit = snnull_tx,
    .ndo_do_ioctl  = snnull_ioctl,
    .ndo_get_stats = snnull_stats,
};

```

Each callback:

Function	Purpose
<code>ndo_open()</code>	Called when interface is brought up
<code>ndo_stop()</code>	Called when interface is shut down
<code>ndo_start_xmit()</code>	Called when a packet needs to be transmitted
<code>ndo_get_stats()</code>	Provides traffic statistics
<code>ndo_do_ioctl()</code>	Handles custom ioctl() requests

◊ 8. Device Deregistration — `unregister_netdev()`

In your module's cleanup function:

```
for (i = 0; i < 2; i++) {  
    if (snull_devs[i]) {  
        unregister_netdev(snull_devs[i]);  
        free_netdev(snull_devs[i]);  
    }  
}
```

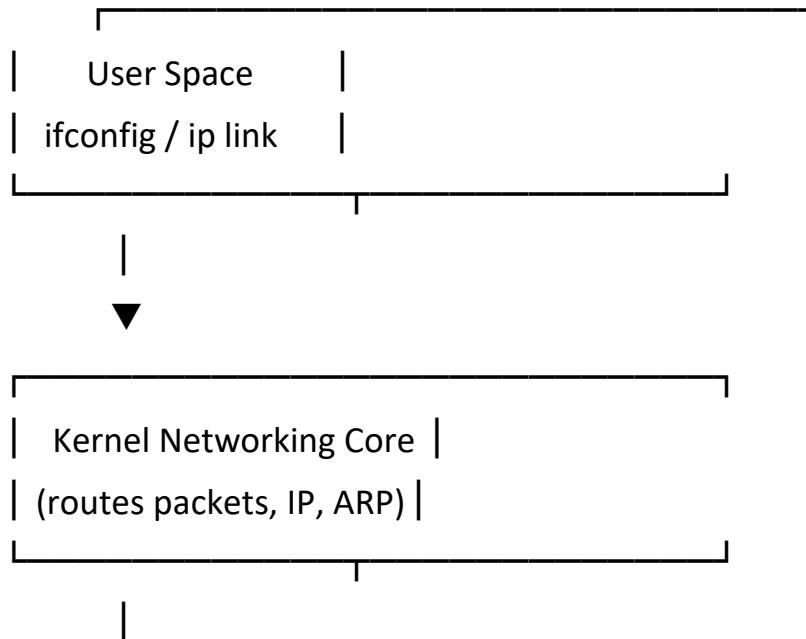
This:

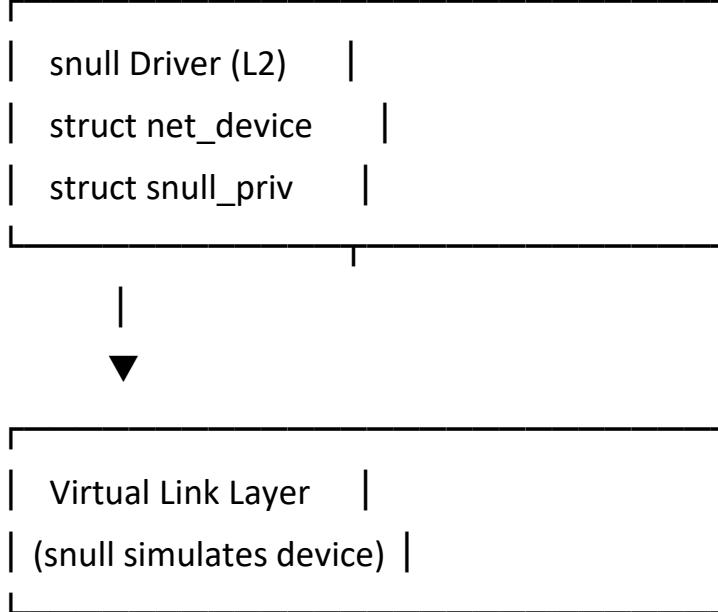
- Unregisters the device from the kernel.
- Frees memory allocated for it.

⌚ Summary Flow — Step-by-Step

Step	Function	Description
1	<code>alloc_netdev()</code>	Allocate memory for <code>net_device</code> + private struct
2	<code>snull_init()</code>	Initialize <code>net_device</code> fields (ops, addr, flags, etc.)
3	<code>register_netdev()</code>	Register the interface with kernel networking core
4	<code>ndo_open()</code>	Called when interface brought up
5	<code>ndo_start_xmit()</code>	Called when packets are sent
6	<code>unregister_netdev()</code>	Remove interface during module removal
7	<code>free_netdev()</code>	Free memory

◊ 9. Visualization





◊ Example Summary Code (simplified)

```

static struct net_device *snull_devs[2];

int snull_init_module(void)
{
    int i, result = 0;
    for (i = 0; i < 2; i++) {
        snull_devs[i] = alloc_netdev(sizeof(struct snull_priv), "sn%d", snull_init);
        if (!snull_devs[i])
            return -ENOMEM;
        result = register_netdev(snull_devs[i]);
        if (result)
            printk(KERN_ERR "snull: error %i registering %s\n", result, snull_devs[i]->name);
    }
    return 0;
}

void snull_cleanup(void)
{
    int i;

```

```

for (i = 0; i < 2; i++) {
    if (snnull_devs[i]) {
        unregister_netdev(snnull_devs[i]);
        free_netdev(snnull_devs[i]);
    }
}
}

```

◊ [10. After Loading the Module](#)

When you run:

`sudo insmod snull.ko`

`ifconfig -a`

You'll see:

`sn0: flags=4098<Broadcast,Multicast> mtu 1500`

`sn1: flags=4098<Broadcast,Multicast> mtu 1500`

These are your **two virtual Ethernet interfaces**, fully managed by your snull driver.

[TL;DR Summary](#)

Stage	Function	Description
Allocate device	<code>alloc_netdev() / alloc_etherdev()</code>	Allocate and name the interface
Initialize	<code>snnull_init()</code>	Fill <code>net_device</code> fields and private data
Register	<code>register_netdev()</code>	Make visible to kernel and userspace
Define behavior	<code>struct net_device_ops</code>	Define how it sends/receives packets
Cleanup	<code>unregister_netdev() + free_netdev()</code>	Remove and free the device

Would you like me to now show the **transmit/receive path (`ndo_start_xmit` + packet flow between `sn0` and `sn1`)** next? That's the next stage after registration — how snull actually simulates packet transmission.

Simple code

Wednesday, October 29, 2025 2:26 PM

```
#include <linux/module.h>
#include <linux/netdevice.h>
#include <linux/etherdevice.h>

#define DRV_NAME "snull"

static struct net_device *snull_devs[2]; // Two virtual interfaces

//Open function — called when 'ifconfig up' is run

static int snull_open(struct net_device *dev)
{
    netif_start_queue(dev);
    pr_info("%s: device opened\n", dev->name);
    return 0;
}

// Stop function — called when 'ifconfig down' is run

static int snull_stop(struct net_device *dev)
{
    netif_stop_queue(dev);
    pr_info("%s: device stopped\n", dev->name);
    return 0;
}

// Start Xmit — required placeholder for registration

static netdev_tx_t snull_start_xmit(struct sk_buff *skb, struct net_device *dev)
{
```

```

pr_info("%s: dropping packet (dummy transmit)\n", dev->name);
dev_kfree_skb(skb);
return NETDEV_TX_OK;
}

// net_device_ops structure

static const struct net_device_ops snull_netdev_ops = {
    .ndo_open     = snull_open,
    .ndo_stop     = snull_stop,
    .ndo_start_xmit = snull_start_xmit,
};

// Device Initialization (called for each sn0/sn1)

static void snull_setup(struct net_device *dev)
{
    ether_setup(dev); // Fill in Ethernet defaults
    dev->netdev_ops = &snull_netdev_ops;
    dev->flags |= IFF_NOARP; // No ARP (simplified)
    strcpy(dev->name, "sn%d"); // Names: sn0, sn1
}

// Module Initialization — Register both interfaces

static int __init snull_init(void)
{
    int i, result;

    for (i = 0; i < 2; i++) {
        snull_devs[i] = alloc_netdev(0, "sn%d", NET_NAME_UNKNOWN, snull_setup);
        if (!snnull_devs[i])
            return -ENOMEM;
    }
}

```

```

result = register_netdev(snull_devs[i]);
if (result) {
    pr_err("snull: error %i registering device %s\n", result, snull_devs[i]->name);
    free_netdev(snull_devs[i]);
    return result;
}
pr_info("snull: registered device %s\n", snull_devs[i]->name);
}

pr_info("snull module loaded.\n");
return 0;
}

// Module Exit — Unregister both interfaces

static void __exit snull_cleanup(void)
{
for (int i = 0; i < 2; i++) {
    if (snl_devs[i]) {
        unregister_netdev(snl_devs[i]);
        free_netdev(snl_devs[i]);
        pr_info("snull: unregistered device %d\n", i);
    }
}
pr_info("snull module unloaded.\n");
}

module_init(snull_init);
module_exit(snull_cleanup);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Example");
MODULE_DESCRIPTION("Minimal snull-like network driver (registration only)");

```

Let's go step by step and connect **theory + snull.c code + kernel flow** ↗

💡 1. Where open() and close() Are Called

When you registered your network device in `snull_init_module()`, you did this:
`register_netdev(dev);`

That registration tells the kernel:

“Here is a new network interface. Use these callbacks for its operations.”

During initialization, you set:

```
dev->open  = snull_open;  
dev->stop  = snull_release;
```

These function pointers are part of `struct net_device`.

Now, the kernel will call them **automatically** when you run user-space commands like `ifconfig` or `ip link`.

🌐 When open() is called

When you execute:

```
sudo ifconfig sn0 up
```

or

```
sudo ip link set sn0 up
```

→ The kernel calls:

```
snull_open(struct net_device *dev)
```

That's the “device enable” callback.

It's responsible for:

- starting the transmission queue,
- initializing interrupts or timers,
- marking the interface as active.

In `snull`, you'll see:

```
static int snull_open(struct net_device *dev)  
{  
    memcpy(dev->dev_addr, "\0SNUL0", ETH_ALEN);  
    netif_start_queue(dev); // <-- enables packet transmission  
    return 0;  
}
```

 When close() (or stop()) is called

When you run:

```
sudo ifconfig sn0 down
```

or

```
sudo ip link set sn0 down
```

→ The kernel calls:

```
snull_release(struct net_device *dev)
```

In snull, that looks like:

```
static int snull_release(struct net_device *dev)
{
    netif_stop_queue(dev); // <-- disables packet transmission
    return 0;
}
```

This function should:

- stop the TX/RX queue,
- disable interrupts,
- release resources.

2. What Are netif_start_queue() and netif_stop_queue()?

These are **queue control helpers** for the kernel's network scheduler.

Every net_device has a transmission queue (struct netdev_queue), which handles outgoing packets.

 netif_start_queue(dev);

- Tells the kernel:
“The interface is ready to send packets.”
- It enables the queue so the kernel networking stack can call your hard_start_xmit() function to transmit packets.

So when you “bring up” an interface, you *start* the queue — allowing upper layers (like TCP/IP) to send frames.

 netif_stop_queue(dev);

- Tells the kernel:
“Stop sending me packets for transmission — I’m busy.”

- It's usually used when:
- The transmit buffer is full.
- A hardware error occurs.
- The driver is being shut down.

If the queue is stopped, the kernel won't call your snull_tx() function.

Example Inside snull_tx()

In snull, when a packet is transmitted:

```
static int snull_tx(struct sk_buff *skb, struct net_device *dev)
{
    struct snull_priv *priv = netdev_priv(dev);

    ...
    netif_stop_queue(dev); // stop until transmission is "done"
    ...
    snull_hw_tx(skb, dev); // simulate sending to the other interface
    ...
    netif_start_queue(dev); // resume queue once done
    return 0;
}
```

So snull temporarily stops the queue while it processes a packet and restarts it once it's done — mimicking hardware transmit completion.

3. TL;DR Summary

Function	When Called	Purpose
snnull_open()	when you run ifconfig sn0 up	Enable device, set MAC, start queue
snnull_release()	when you run ifconfig sn0 down	Stop queue, clean up
netif_start_queue()	Called inside open()	Tells kernel "ready to send"
netif_stop_queue()	Called inside stop() or during TX	Tells kernel "don't send now"

In short:

- open() ↔ ip link set dev up
- stop() ↔ ip link set dev down
- netif_start_queue() ↔ “transmission ON”
- netif_stop_queue() ↔ “transmission OFF”

Would you like me to show the **exact path in the kernel source** (how ifconfig up leads to dev->open() being called internally)? That's a nice deeper trace through net/core/dev.c.

